Comparing the Performance of String Operations Across Programming Languages

University of Oulu
Faculty of Information Technology and Electrical Engineering / Information Processing Science
Master's Thesis
Niko Pelkonen
20.12.2019

Abstract

In this thesis, the performance of string operations are compared across programming languages. Handling strings effectively is important especially when performance is a crucial factor and large string sizes may emerge. Common examples where large string sizes emerge are during digitalization of a product, reading string data from a database, reading and handling large CSV-files and Excel-files, converting file format to another file format (e.g. CSV to Excel and vice versa), and reading and handling a DOM-tree of a website.

There has been a lot of corresponding research where programming languages are benchmarked, but none of them focus directly on string operations. The main goal of this thesis is to fill this gap in literature and try to find out which programming languages have the best results on string operations in terms of execution time and memory (maximum RSS) usage.

The test environment was formed by creating randomly generated string files with sizes varying from ten thousand characters to 100 million characters. The generated characters were ‘a’, ‘b’, and ‘ ‘ (whitespace character). The programming languages selected for this thesis were Python, C, C++, Java, Perl, Ruby, Go, and Swift.

Go seemed to be the most effective language in execution times, although it was not the fastest in many operations. C used very little memory, but only five operations were implemented in it. Every operation was implemented in Python, and it used additional memory to loading the string file in only one operation, which was sorting a string. Swift had quite bad results, and this could be caused by the Linux version of Swift that was used. In regular expressions, Perl and C++ were overwhelmingly effective. Java used the most memory in every operation.

Keywords
programming languages, comparison, string operations

Supervisors
Ph.D., University lecturer, Ari Vesanen
Ph.D., University lecturer, Antti Siirtola

Foreword

I have a bad habit of taking too much things to handle at the same time, and that happened also during this thesis, which resulted in yet another busy and stressful period of time while writing this thesis. Nevertheless, writing this thesis has been truly fascinating and inspiring; picking a subject of interest, creating the test environments, seeing interesting and surprising results, reporting the results thrillingly, and discussing about the results with other people has been a privilege that I'm thankful for.

I remain grateful to many people who have helped me during this journey. I'd like to thank my supervisors, University lecturers Ari Vesanen and Antti Siirtola, who gave great guidance throughout the process. Special thanks for my friends Marko Pulkkinen, who helped me a lot in choosing my topic, and Juho Junnila for not only helping out with this thesis, but also for all these amusing years at the University of Oulu we've gone through. Thank you also for my family for having had patience and understanding with me being a lot away due to my duties.

Niko Pelkonen

Oulu, October 30, 2019

Contents

1. Introduction

This introductory chapter is divided into four subchapters. The first subchapter discusses the motivation of this study. The second subchapter introduces strings and why they are relevant in this study. The third subchapter presents the research questions for this thesis and the research methods used. The last subchapter presents the structure of this thesis.

1.1 Motivation

The need to handle strings of great size and the performance issues that are brought within them are common matters in programming. A program should be able to handle strings of great size, especially if performance is an important requirement. If the size of the strings is not considered beforehand, large strings may become a burden for developers. Therefore, sometimes developers need to consider which language would be the best option for handling large strings effectively.

Programming languages have built-in string operations, but which of those methods and languages itself are efficient enough in terms of performance? It is also possible that some of the built-in methods or functions perform efficiently on small-sized strings, but as the size of the string grows, the performance drops. There hasn't been any research done on the very subject; naturally, a lot of research has been done on performance comparisons across programming languages, but string operations specifically has been left out so far. This research aims to fill that gap in the literature and find out which programming languages tend to perform the best on varying string operations.

1.2 Strings

Strings in programming are typically sequences of characters. How they are declared, handled, stored in memory etc., is up to a programming language, and there are variations across programming languages. (Busbee, 2009) There are many situations in which large strings may emerge, including:

- Reading and handling large Excel-files (there may be hundreds of thousands of rows and tens of columns),
- Reading and handling large CSV-files,
- Converting a file format to another file format (e.g. converting Excel-file to CSV-file and vice versa),
- Reading string data from database,
- Reading and handling data from large files in order to digitize a product, and
- Reading and handling DOM from a website (e.g. if a website doesn't provide an API (application programming interface) for getting the data it presents, and the site includes some data that must be used, the data must be read from the DOM of the site).

DOM (Document Object Model) is an application programming interface (API) that represents the logical structure of the HTML-elements (e.g. head, body, div, tr) on a

website as XML and specifies the website's interface. DOM provides a standard programming interface which developer can use to manipulate elements and navigate the structure. (W3C, 2000) It is presented as a tree-structure, making it easy to interpret. DOM is returned as a string (which might be massive), and string operations are performed in it in order to get the data that is needed. For example, first thing could be to remove all other than div-elements, and from these div-elements one could search for elements with a specific id (an identifier for elements). This id could be, e.g., for elements that include a person's name. These names of persons could be then saved in to another variable, which could be, e.g., sorted alphabetically.

A key point why strings need to be handled effectively is because data is often received as a string, and therefore it needs to be handled as a string. For example, data received from a CSV-file is a string (e.g. columns separated with a comma and rows separated with a semicolon), and the same goes for DOM of a website. Excel-files usually give user freedom to write whatever user wants, so there are usually strings in Excel-files, too. Data stored in database may also include a large string. In addition, when Excel-file needs to be converted into CSV-file and vice versa, the data needs to be converted and it has to be handled as string.

An example of working with a string returned from a CSV-file could be to first get data from each row and column. As mentioned earlier, rows and columns are often separated with characters like colons and semicolons. Thus, the first string operation would be to split the string with colons and semicolons. Then, regular expressions could be used to check that substring doesn't include any illegal characters; if it does, those could be removed or modified. Checking if substring equals a fixed value, as well as concatenating some value to a substring, are also common string operations.

1.3 Research questions and methods

The current literature doesn't include the exact subject of this thesis; which programming languages are the most efficient on string operations. This thesis aims to answer this question. Thus, the research questions are:

1. Which programming languages have the shortest execution times on string operations?
2. Which programming languages use the least memory on string operations?

The research method used in this thesis is experimental research. There are various different types of experimental research, and benchmarking is the type of experimental research used in this thesis. Benchmarking is a common approach in this kind of research where different technologies are being compared, and it fits this thesis well.

1.4 Structure

This thesis has seven chapters, introduction included. Chapter 2 presents the current literature on the differences of programming languages; there has been lots of studies in which programming languages are being compared, but none of them focus on string operations only. Chapter 3 introduces the research methodologies, the sample strings used to compare the languages, the programming languages selected for the thesis (how were they selected, how they handle strings, etc.), what string operations were selected for the thesis, and what problems emerged with measuring memory usages. Chapter 4

presents the results of the comparisons as simple graphs implemented in R. Each string operation has its own subchapter in which the graphs are shown and the results are discussed. Chapter 5 contains discussion of the results; what operations were the most demanding (in terms of execution time and memory usage), what programming languages performed the best and what performed the worst, are there any differences between compiled and scripting languages, and so on. Chapter 6 is the last chapter, and it presents the conclusion of the thesis as well as limitations and suggestions for future research.

2. Background

The comparisons of string operations performance across programming languages hasn't been that researched; some studies have included some string operations when programming languages were being compared, but there hasn't been any explicit study on string operations only. However, there are plenty of papers in which the performance of programming languages are compared in general.

This chapter presents the current literature on the subject. The references mostly study the differences on memory consumptions and execution times across programming languages, which is also studied in this thesis. There are some languages and benchmark measurements (e.g. energy consumption) that are discussed in this chapter that are not directly part of the scope of this thesis. Nevertheless, they were added as viewpoints on how programming languages may be compared.

Chapter 2.1 consists of literature on execution time and memory usage comparisons, and chapter 2.2 consists of literature on lines of code and reliability of languages.

2.1 Execution time and memory usage comparisons across programming languages

In a study by Prechelt (2000), programmers implemented programs on the same programming problem on seven different languages, which were C, C++, Java, Perl, Python, Rexx and Tcl. The program had to load a dictionary of over 70 000 words into memory, and then read "telephone numbers" from another file. Those numbers had to be converted into word sequences, and the result was printed. Not surprisingly, implementing the program on scripting languages (Perl, Python, Rexx and Tcl) took the least time (especially on Perl and Python); the research shows that implementations in Perl, Python, Rexx and Tcl took under half the time it took on C, C++ and Java. Java and C++ programs took the most time to implement, but as for Java, it should be noted that the developers on the study were inexperienced Java programmers. Java programs used three to four, and scripting languages about two times more memory than programs written in C and C++. C and C++ were also about three to four times faster than Java, and five to ten times faster than scripting languages. (Prechelt, 2000) This paper is a great background source, because it has similarities to my study. The similar languages were C, C++, Java, Perl, and Python, and the paper studied execution times and memory usages, which is also studied in my thesis. Also, the paper used a certain data set (z0, which is empty) to measure the runtime for dictionary load time only, and in my study I had to take file loading into account in memory usage tests (in execution time tests this wasn't an issue, because they were done using the system clock in languages). There are some differences between my study and Prechelt's. In my study, I develop the programs myself, but in Prechelt's study, the programmers were other than the author itself; for Java, C, and C++ programs the programmers were computer science master students, and for Perl, Python, Rexx, and Tcl programs the programmers were volunteers for the study. Another difference is that in Prechelt's study only one kind of program was measured, whereas in my study I measure different programs (varying operations).

Fourment and Gillings (2008) found out that C and C++ programs used the least memory and they also had the shortest execution times. Perl and Python were pointed out to be flexible languages that required little amount of work, and Java and C# were like a compromise between these two groups. In this study, different kinds of programs were implemented to compare the languages. One type of program was to parse an output of a BLAST (Basic Local Alignment Search Tool) result, and from these implementations, the slowest languages were Python and C#, and the fastest were C and C++. Other programs were global alignment program, and Neighbor-Joining program, and in these two, Perl and Python were clearly the slowest. The results for C and C++ were very much alike; overall, they were the fastest programs and required the least memory. Python had clearly worse results than Perl for I/O operations. In addition, C# consumed more memory than Java for holding strings in memory. (Fourment & Gillings, 2008) Compared to my thesis, this paper also studied execution times and memory usages, there were similar languages (namely, every language except for C#, which had to be excluded from my thesis), and the languages were measured against varying programming tasks. One difference was that the paper compared operating systems (no clear evidence was found whether Windows or Linux would be faster operating system) whereas my study doesn't. Also, the programs in Fourment & Gillings' paper were considerably bigger; in my study, where the programming languages' string operations are compared, an operation takes usually only one line of code.

Aruoba and Fernandez-Villaverde (2014) studied execution times on programming languages commonly used in economics. These languages were C++, Fortran, Java, Julia, Python, Matlab, Mathematica, and R. In these languages, the stochastic neoclassical growth model was implemented. They found out that the fastest languages in their study were C++ and Fortran, C++ being slightly faster. Julia was quite effective language, being only 2.64 to 2.70 times slower than C++. Python was found to be remarkably slow; with Pypy implementation, which is a virtual machine replacement using just-in-time compiler, it was over 40 times slower than C++, and with the traditional CPython implementation, it was 155 to 269 times slower than C++. Java was over 2 times slower than C++. (Aruoba & Fernandez-Villaverde, 2014) This study points out great background information for my thesis, because it studied execution times of varying programming languages, which is also part of this thesis. The same languages between this study and my thesis were C++, Java, and Python. The difference is that the authors implemented only one program on every language, whereas my thesis compares various operations.

Couto, Pereira, Ribeiro, Rua, and Saraiva (2017) carried out another study using 13 benchmark problems from the Computer Language Benchmarks Game (CLBG) in order to study the execution times and energy consumptions of programming languages. The rise of non-wired computer devices has led to energy consumption being a bottleneck on building computers and their softwares. The languages chosen for the study are C, C#, Fortran, Go, Java, JRuby, Lua, OCaml, Perl and Racket. Intel's Running Average Power Limit (RAPL) tool was used for measuring energy consumption. According to the authors of the study, one of the hot questions that tends to rise on the subject has been whether a program with high execution times will also mean it is energy efficient. The overall results for the 13 benchmark problems show that for both energy consumption and execution time C was the best with Java following as the second. C# was 5th in energy consumption, consuming 2.21 times more energy than C, and 3rd in execution time, being 2.44 times slower than C. Perl used the most energy out of the 10 languages, consuming 84.89 times more energy than C. Perl was also the second most slow as it was 68.45 times slower than C. In addition, it was noted that energy

consumption isn't always directly proportional to execution time, as almost every task had languages where energy consumption and execution time didn't behave in the same way. An interesting finding was also that Lua was 12% slower than Perl, but at the same time, it was 53% greener. Go was a pretty solid performer in each test. (Couto et al., 2017) This study had also similarities with my thesis. It included benchmarking of programming languages, and some of the languages studied were the same as in my thesis: C, Go, Java and Perl. In addition, execution time was studied. What was different, though, was that this paper studied energy consumption, whereas my study studies memory consumption in addition to execution times. Also, this study used 13 different CLBG programs to compare the languages, but my thesis uses strings of varying sizes.

Energy, time and memory consumption across programming languages were studied by Pereira et al. (2017), who also used CLBG in their study. The results for energy and time consumption suggested that the best performer was C, followed by (from best to worst) Rust, C++, Ada, and Java. It was noticed in the study that execution time doesn't necessarily affect energy consumption directly; for example, Fortran consumed second least memory for one CLBG problem, but at the same time, it was only 8th in execution time. On average, the least memory was used by Pascal (66Mb), Go (69Mb), C (77Mb), Fortran (82Mb), and C++ (88Mb), whereas the most memory was used by JRuby (1309Mb), Dart (570Mb), Erlang (475Mb), Lua (444Mb), and Perl (437Mb). (Pereira et al., 2017)

Rosetta Code is a repository which collects programming solutions to varying tasks in different languages. Anyone can log in to the repository and add a new task or a solution, either to seek or provide help with programming subjects. ("Rosetta Code", n.d.) Nanz and Furia (2015) used Rosetta Code in order to find out about differences in programming languages. They were looking for differences in lines of code, sizes of executables, runtime performances, memory usage efficiencies, and runtime failures. The study analyzed over 7,000 solutions to over 700 different tasks in 8 languages. The languages selected for the study were C, Go, C#, Java, F#, Haskell, Python, and Ruby. The results of the study suggest many kinds of matters. F#, Haskell, Python and Ruby enabled writing more concise code than C, Go, C# and Java. Executables compiled into bytecode were smaller than the ones compiled into native machine code. For both runtime and memory usage, C had the best results, and Go was the second. However, the differences on runtime reduced on moderate sized inputs. In addition, compiled strongly-typed languages were able to catch more errors at compile time than interpreted or weakly-typed languages. (Nanz & Furia, 2015) I think this study is reliable because of the large sampling size (there were over 7,000 solutions analyzed). This study has also valuable background information for my thesis, because there were similar languages (C, Go, Java, Python, and Ruby) studied, and execution times and memory usages were studied, too. The key difference between Nanz & Furia's paper and my study is that they analyzed over 7,000 programs found at Rosetta Code, whereas I develop the programs myself. Another difference is that the programs analysed in Nanz & Furia's paper were much bigger programs than my programs, which are mostly programs consisting of one string operation (excluding, e.g., reading the string file).

Cesarini, Pappalardo and Santoro (2008) studied the differences of programming languages by implementing an IMAP (Internet Message Access Protocol) client on each programming language. The programming languages they selected for their study were Erlang, C#, Java, Python and Ruby. The metrics for comparison they used were SLOC (number of source lines of code), memory consumption (the space cost of the library), execution time, and functionality of primitives (analyses the functionalities of primitives

provided by the IMAP libraries). SLOC was highest for Ruby (1,612) and, not surprisingly, smallest for Python (472). For C# SLOC was 1,089, and for Java it couldn't be measured. The amount of memory required for Java was highest, as it required 14MB for code, 190MB for class library, and a total of 213MB. Python required only 4,4MB for code, 1,6MB for library, and 6,7MB in total. The total of memory required for Ruby was 15MB and for C# it was 19MB. As for execution times, Ruby's performance was the worst, and the performance for C# was the second worst. Python seemed to have the best performance, but it should be noted that Python didn't parse IMAP server responses. Overall, for the client-side IMAP protocol implementations, Ruby had the worst results, whereas the best results were calculated for Python. (Cesarini et al., 2008) The study consisted of developing only one program on all of the selected languages, which is a bit lacking. Programming languages have different kinds of results on different types of programs. For example, some Python program might have higher execution time than a similar Perl program, and vice versa. Therefore, this study would be more reliable if it had consisted of developing more programs, because now the results might not be generalizable. However, it is a nice addition to the background information, especially because it included execution times for Java, Python and Ruby, which are also part of this thesis.

JIT-compiled (just-in-time) dynamic languages have started to challenge compiled languages as they have began improving their performance. Fortran, C++, Java (compiled languages) were compared to Python (with Numba-library), Matlab, and Julia (interpreted languages) by Eichhorn, Cano, McLean and Anderl (2018) in their research. The languages were evaluated with four different problems relevant in astrodynamics. The research found out various things. Julia was faster than Java in every test, and even faster than Fortran and C++ in one test. Matlab was the slowest in every test. Python+Numba was also a worthy challenger for Fortran, C++, and Java in runtime. However, C++ and Fortran still seemed to be the best alternatives for astrodynamics, where high performance is needed. (Eichhorn et al., 2018)

In a study by Alomari, El Halimi, Sivaprasad and Pandit (2015), C#, C++, Python, Java, VB and PHP were compared by using BFS (breadth-first search), DFS (depth-first search), Kruskal's algorithm, as well as reading and writing operations on a MySQL database. Tests for DFS, BFS and Kruskal's algorithm were used to evaluate programs execution times and memory consumptions, and database tests were used to evaluate execution times only. For DFS, BFS and Kruskal's algorithm, C++ had the best results. A rather interesting finding was that for DFS, Python was the second best in execution time, but for BFS, it was clearly the worst. Java was the worst performer on memory consumption on tests for DFS, BFS and Kruskal's algorithm. C# was pretty solid performer in both memory consumption and execution time. Java seemed to be faster than C# in writing operations, but C# was faster in reading operations. (Alomari et al., 2015)

2.2 Comparisons on SLOC and code quality across languages

Although SLOC (source lines of code) and code quality are not part of this study, this chapter presents two researches on those subjects; they make great background references, because the researches consist of comparisons across programming languages.

Typing of programming languages may have an impact on program's ability to produce erroneous results. This was supported by Spinellis, Karakoidas and Louridas (2012)

who studied the very subject. They selected C, C++, C#, Haskell, Java, JavaScript, PHP, Perl, Python and Ruby on their study, and they used 14 different tasks from Rosetta Code to evaluate the languages. Each language 136 implementations were tested using a "fuzzing" tool that introduced random perturbations into the code. The perturbations were such that could happen for anyone (e.g. mistyping). The results proved that languages with strong typing (Java, Haskell, C++) were less likely to run successfully and/or produce erroneous results than languages with weak or dynamic typing (Ruby, Python, Perl, PHP, and JavaScript). The results for C were somewhere in the middle. PHP had the worst results, as it produced an erroneous output in 36% of the test cases, which was clearly the worst rate, and it ran successfully in 40% of the cases. The lowest error rates were for C++ (8%), C# (10%), C (10%), and Java (10%). High error rate means, in this case, that mistypings will slip more likely undetected into production code. (Spinellis et al., 2012) The results of the study clearly suggest that languages with strong typing are safer than those with weak or dynamic typing. The study is also reliable, because each language had approximately 14 solutions (not every task had been implemented on some languages), which is plenty. Also, the fuzzing tool used made random perturbations, and not just in some specific parts of the code.

In evaluating source code quality, Cyclomatic Complexity (CC) is often used, as well as Source Lines of Code (SLOC). However, it has been claimed that CC is redundant. Landman, Serebrenik and Vinju (2014) studied a corpus of almost 18 million methods in about 13 000 Java projects. They found out various things, but most importantly, there is no strong linear correlation between CC and SLOC of Java methods; thus, they suggest that CC is not redundant for evaluation source code quality. (Landman, Serebrenik & Vinju, 2014) This study was revisited in 2015 with the addition of involving C functions; 19 000 open-source Java projects from Sourcerer and 13 000 C projects from Gentoo distribution were analysed. In this analysis, the results were the same: there was no strong linear correlation between CC and SLOC. (Landman, Serebrenik, Bouwers & Vinju, 2015) In contrast to these two researches, a study by Jay et al. (2009) suggested the opposite. They studied the relationship between CC and lines of code (LOC) on C, C++, and Java by randomly selecting over 1.2 million source files from SourceForge code repository. This study suggested that CC and LOC have a "stable practically perfect linear relationship". They measure the same property, regardless of programmer, language, software process, or code paradigm, and that CC has "no explaratory power of its own". (Jay et al., 2009) This discrepancy between results leave the question whether CC is useful or not unanswered. Neither CC nor SLOC will be used in this study, because the code samples are very concise in this study; a test for some string operation usually takes only one row of code, excluding things like loading the file, loading necessary libraries etc.

3. Methodology

This chapter will discuss the research method used, introduce the strings used for measuring the programming languages, what programming languages are used, and string operations used in the study.

3.1 Research method

This is an experimental research that aims to find out the differences between the selected programming languages' string operations. According to Neuman (2011), with experiments (that are usually artificial, simplified situations about real world), you can find about the causal relationships between the investigated variables. Experimental researches tend to be best suited for issues with a narrow scope (Neuman, 2011) - in my opinion, this makes sense, because in order to carry out experiments that are effective and purposeful, it is only natural that the research objects and questions shouldn't be too wide.

There are a number of different types of experimental methodologies: in-situ, emulation, simulation, and benchmarking (Gustedt, Jeannot & Quinson, 2009). This research is about benchmarking, which is an effective and not that expensive way for performing experimental research. Benchmark essentially means an executable sample of a task domain, and during execution, some characteristic of the sample is measured. With benchmarking, it is easy to identify promising approaches and rule out bad ones. Benchmark studies are also easy to repeat. (Tichy, 1998) Ultimately, the goal in benchmarking is create an output of a program that enables comparison across varying implementations of the same type of program. Therefore, the test setting is documented among the results, so that the results of the benchmark may be compared and a benchmark study may be repeated. (Bouckaert, Gerwen & Moerman, 2011) With benchmarking, one could measure for, e.g., I/O performance or CPU speed (Gustedt, Jeannot & Quinson, 2009). In this research, benchmarking is used to measure memory usage and execution time.

In-situ experiments mean executing some application in a real-world environment. In-situ experiments are useful especially when some complex behavior and/or interaction of operating systems cannot be captured with emulation or simulation. However, in-situ experiments are more about field experiments, and that is not the case in our study, so in-situ experiment isn't a suitable method to be used. In emulation, the goal is to use a set of synthetic experimental conditions, and the environment is modeled. A virtual machine is a well-known example of an emulator. Emulation isn't suitable for this study, because there is no need to use any synthetic experimental conditions. Lastly, in simulation, a part of an environment is focused on and abstracted from the rest of the system, and only a model of an application is executed. This approach is not suitable for my study, because this study isn't about executing a model of an application on a model of an environment. (Gustedt, Jeannot & Quinson, 2009)

According to Nguyen, Deeds-Rubin, Tan & Boehm (2007), size is a highly important attribute of a software product, and SLOC (source lines of code) is the most popular

sizing metric, but it has some drawbacks, which are brought up by Jones (1978): a line of code is something that different people address differently. Some think that comments and data declarations, for example, are counted as lines of code, where others don't. This may become a problem in case there isn't a clear understanding between the persons discussing what is a line of code. Another problem might occur on programs written in high-level languages: a line may be addressed as, e.g., everything between semicolons, or everything on a single line. (Jones, 1978) However, although SLOC is useful to be included in a study, it is not relevant in this thesis, because most of the operations implemented are one-liners (excluding loading libraries, string files etc.).

3.2 Sample strings

The sample strings used in the study are randomly generated strings consisting of three different characters: "a", "b", and " " (whitespace). Characters "a" and "b" had a probability of 35% and whitespace had a probability of 30% to generate. This format was chosen because it is easy to reproduce in the future in case this study needs to be revisited. Also, it is wise to have a whitespace as one character, because it makes the strings more readable. These types of strings which simply consist of three types of characters are sufficient for testing purposes, because the size of the string will be the same regardless of whether the string consists of three different characters or 20 different characters; it will still require the same amount of memory when the file is loaded, and when there are only a few different characters, some algorithms might have lower performances. For example, when searching for specific substrings, e.g. 'abba', if the string consists of more than a couple of characters, it is likely that it will be faster to find matches, because when there are more characters, the program will progress faster as it finds characters that it is not looking for.

There will be five different sizes of strings used in the research, starting from 10,000 characters and increasing up to 100 million characters. The size of a string will always be multiplied by 10; this makes it easy to see the differences across programming languages, as the sizes of the strings steadily increase. Also, the increasing size of the strings point out whether some language is efficient on small sizes but inefficient on large sizes and vice versa. So, the sizes of the strings will be 10,000, 100,000, 1 million, 10 million, and 100 million characters.

3.3 Programming languages

The criteria for selecting a programming languages as a part of this study are: a) the language must be a popular, widely used language, b) the language must run on Linux-based operating systems (because that is used in this thesis), and c) the language must be either a compiled language or a scripting language (e.g. PowerShell scripting doesn't belong to either group), because the study aims to make comparisons not only between all selected languages, but also between these two groups and among the groups. Java doesn't fall into either of these groups, because Java is a hybrid language; first, the compiler converts the code to bytecode, and then this compilation is converted into binary code by the interpreter (Rao, 2015). It was decided that JavaScript and TypeScript will be excluded from this study, because being web-development languages, they differ from other languages in this study (e.g. they run on browser). Also, C# had to be excluded, because it is not supported on Linux-based operating systems.

Stack Overflow along with GitHub are perhaps the best sources for finding out which languages are the most commonly used. Both of these sites also publish annual data on programming languages, and these are used in order to find out the most suitable languages for the thesis. Stack Overflow's annual developer survey pointed out the 25 most popular programming, scripting, and markup languages of 2018 among many other survey results. There were 78,334 respondents on technology popularity survey. (Stack Overflow, 2018). GitHub also published their annual results for various things, including which programming languages have the most contributors, which is used in this thesis ("Projects", 2018). These sources will be used to demonstrate the popularity of the languages selected for this research.

The popularity of the selected programming languages are shown in Table 1. Swift, Go and Perl weren't included in the top 10 of GitHub's popularity measurement, and no further results were published, so their values are marked as "n/a" (GitHub, n.d.).

Table 1. Popularity of the selected programming languages sorted by Stack Overflow popularity. ("Developer Survey Results 2018", 2018; GitHub, n.d.)

Programming language	**Stack Overflow popularity**	**GitHub popularity**
Python	#5	#2
Java	#7	#3
C++	#10	#5
C	#11	#9
Ruby	#13	#10
Swift	#14	n/a
Go	#16	n/a
Perl	#25	n/a

Table 2 shows the compiler versions of the programming languages.

Table 2. Compiler versions of the programming languages.

Language	**Compiler version**
C	8.2.1
C++	8.2.1
Java	OpenJDK 1.8.0_201
Go	1.12 linux/ amd64
Swift	4.2.1
Perl	5.28.1
Ruby	2.5.3p105
Python	3.7.2

Each programming language in this study uses default character encoding. The programming languages selected for the study are:

- C. There is no specific data type for string; in C, a string is an array of characters. Each character of a string has type *char*. (Vahtera, 2003) Strings in C are mutable, unless type *const* is used. An *undefined behavior* results in case a *const* variable is tried to change. (“const type qualifier”, 2018) C doesn’t use any default character encoding (e.g. UTF-8) as such; rather, it groups characters into sets of source characters (alphabets, digits, special characters, and whitespace characters) and execution characters (escape sequence), and it uses them in constructing a program (”Character set of C”, n.d.). This set of characters is basically what ASCII character encoding consists of.
- C++. C++ has a standard library *string* for string operations, and strings are objects consisting of sequences of characters. The class *string* is an instantiation of the class template *basic_string*. (cplusplus, n.d.-a) According to Stroustrup (2013), C-style strings are considered by some to be more efficient than *string* in C++, but *string* doesn’t do as much allocations and deallocations than C-style strings. Strings in C++ are mutable, so they can be changed during execution (cplusplus, n.d.-b). The basic strings in C++ use the same character set as C, but C++ also has a class for wide characters, “wstring” (cplusplus, n.d.-c).
- Java. Strings in Java have a *String* class which represents character strings, and all string literals are implemented as instances of it. Strings are also constant, meaning that they cannot be changed after they are created, but string buffers support mutable strings. (Oracle, n.d.-b) It should be noted that Java Virtual Machine (JVM) increases Java applications’ execution times and memory usages - reasons for this include things like garbage collection (Oracle, n.d.-c), initializing required classes (Normand, 2019), and the fact that JVM itself requires some amount

of memory. Default encoding in Java is determined by JVM during startup and it depends on the operating system and the locale (Oracle, n.d.-a).

- Go. Unicode compliant and UTF-8 encoded, strings in Go are slices of bytes (Ramanathan, 2017). Like in Java, strings are immutable; once created, it's not possible to change them (Pike, 2013a). By default, strings in Go are simply byte arrays, but the source code needs to use UTF-8 encoding (Pike, 2013b).
- Perl. In Perl, all values, including strings, are simply stored as variables, which the interpreter will then handle. Perl has three data types, and they are used to describe the way the data is organized. These data types include scalars (for a single value), arrays (multiple scalars), and hashes (pairs of scalars). (Lerner, 2002) Variables in Perl are mutable, but they may be declared as constants. A constant may, however, be overrode with a scalar. (Wall, Christiansen & Orwant, 2000) A string in Perl is stored, if possible, as single-byte character codes that use native character encoding. If it's not possible, UTF-8 will be used instead. (Wainwright, 2005)
- Ruby. Like Java, Ruby has a *String* class, and all strings are objects of this class. Strings in Ruby are textual characters that are stored as numbers in computer's memory, and every character has an ASCII value. Any string that is written into code (and not received from another source, e.g. from user input) is called a string literal. (Cooper, 2009) Strings are mutable in Ruby. A string may be declared as a constant, but if a constant is tried to change, a warning is generated instead of an error. (Ruby user's guide, n.d.) The default encoding in Ruby is UTF-8 ("class Encoding", n.d.).
- Swift. In Swift, a string is a series of characters, which are represented by type *String*. There are many ways to access a *String* type (e.g. as a collection of *Character* types), and the mutability of a string is defined explicitly. (Swift, 2018) Swift (version 4, which is used in this thesis) uses ASCII encoding if possible, otherwise it uses UTF-16 encoding (Ilseman, 2019).
- Python. Every variable in Python is treated as an object, and objects have an identity, a type, and a value. Identities and types are both immutable, but values can change. (Python Reference Manual, n.d.) The default encoding in Python is UTF-8 (The Python Wiki, n.d.).

Compiled languages will be analyzed against scripting languages in this thesis. The languages are also analyzed among their group (compiled/scripting languages), e.g. comparison between Go and Java. Compiled languages are C, C++, Swift, and Go, and scripting languages are Python, Ruby, and Perl. Java is a hybrid language and doesn't fall into either of these groups.

3.4 String operations

The string operations were selected by going through built-in methods and functions of programming languages and picking up some of the most common string operations that would be beneficial to study. In addition, each operation had to be implementable (with a built-in method or function) in at least five languages. The operations selected for the study are:

- Concatenate to string,
- Uppercase a string,
- String equality (check whether two strings are equal),
- Duplicate a string,

- Reverse a string,
- Find the first index of a substring,
- Sort a string by ascending/descending order,
- Regular expressions, and
- Replace all substrings with another substring

Table 3 shows which operations are implemented on each of the languages.

Table 3. List of string operations used and whether operations are built-in methods or functions in languages (grey=is built-in, white=is not built-in).

	C	**C++**	**Java**	**Swift**	**Go**	**Perl**	**Ruby**	**Python**
Concat								
Upper								
Equals								
Strdup								
Strrev								
Strstr								
Replace								
Sort								
Regex								

The reason why some operations weren't implemented was because there wasn't a built-in method or function for them. The scope of this thesis was to study operations that are built-in methods or functions in languages; naturally, every operation could have been implemented on every language, but it was decided that it is better and safer for the sake of meaningful and repeatable results that I, as the author, did not develop any operations that were not built-in (small exceptions are mentioned in more detail in the chapters). Take replacing all substrings with another substring in Perl for example: there is no built-in function or method for it, but it could be implemented. However, it can be implemented in numerous different ways, which could have varying results, so it is simpler and more reliable to use built-in functions and methods to avoid misleading or unreliable results. In addition, the study is more easily repeatable when only built-in functions and methods are used.

3.5 Issues with measuring memory usage and execution times

There were some issues with measuring the performances of the programming languages. Perhaps the biggest problem is that memory usage is difficult to measure properly when the program also needs to load the text file containing the string used in

the operations. For example, Java used 34.5 Mb of memory in loading a string file of one million characters, and in chapter 4.2 where the programs had to replace substrings with another substring, Java used 42.6 Mb with the same string size; this result includes the amount of memory Java used for loading the string. In execution time this is not an issue because execution time is done with system clock in each program.

There are various ways to measure memory, so an issue in itself is what memory should be measured. In this thesis, I decided to focus on maximum resident set (or real-memory) size (max RSS), which calculates how many working segment and code segment pages there are in memory (“IBM Knowledge Center”, n.d.).

Another issue is that the values for both max RSS and execution time are not static; the values slightly change on each run. The operating systems always perform some operations and run some programs on the background, and this has an impact on the results. Therefore, I used a script that runs a program 100 times and on each execution the result (execution time or max RSS) is written on a text file. After this, the average value is calculated from this text file. Of course, these steps are implemented so that the results are not affected.

4. Results

This chapter presents the results for all programming languages.

Both execution times and maximum RSS usages were measured by loading the text file and running the operation 100 times. After 100 runs the average values were calculated. It should be noted that loading the text file increased maximum RSS usage on some languages more than others. This didn't affect execution times, because they were measured by getting the current time after and before the operation, after which the execution time was calculated by subtracting the first value from the second one. An example of how execution time was calculated on Python:

```
start = time.time()
string.join('a')
end = time.time()
duration = end-start
```

Maximum RSS usage was measured using Unix's time-library. The measured program was run 100 times, and on each run the output was parsed and the value for maximum RSS usage was written into text file. This text file was then used to get the average value. An example of measuring the average maximum RSS usage of a Python program:

```
niko:Python $ for i in `seq 100`; do /usr/bin/time -v python3
programs-python.py 2>&1 | grep "Maximum resident set size (kbytes)" |
awk '{print $6}' >> times-file.txt; done
```

Execution times were measured using a similar loop, with the exception that there wasn't any output that needed parsing, because the program itself wrote the output into the text file:

```
niko:Python $ for i in `seq 100`; do python3 programs-python.py; done
```

Appendix D shows the libraries and methods used for execution times.

Naturally, different computers, processors etc. produce different results on execution times and memory usages. In this thesis, the following hardware was used:

- Linux (Fedora 27) 64-bit operating system,
- Intel® Core™ i5-7200U CPU @ 2.50GHz × 4 processor,
- GNOME version 3.30.2,
- 7,7 GiB memory,
- 234,1 Gt SSD drive.

There were five different regular expressions that were tested on all languages. The regular expressions implemented include the most important combinations that should be tested. The reason for choosing five regular expressions in this thesis was because they show different results; some of them are more demanding in terms of execution time, and the results across languages simply have variance (excluding C++ and Perl, which were overwhelmingly fast). For example, in Go, regular expressions 1 and 2 were

executed really fast, but in regular expressions 3, 4, and 5, the execution times increased remarkably. The regular expressions are explained more in detail in their respective chapters.

Go consumed very little memory up to 1 million character strings, but on string sizes of 10 million and 100 million characters memory usage increased rapidly. Go consumed the third most memory on almost every test, and it mainly outdone only Swift and Java. Swift took almost 100 MB (megabytes) of memory by just loading the smallest string size (it is unclear whether it was due to Linux version of Swift that was used), and the same result for Java was 26 MB (OpenJDK had an impact on memory usage). On the contrary, the lowest memory usages on loading the smallest string size were for C (1.3 MB) and for Go (1.9 MB). Java consumed most memory on strings of 100 million characters in every operation. C++ and C had solid performances on memory usages, with the exception of duplicating a string, in which Ruby and Perl had better results on memory usage. In string duplication, C++ and C had surprisingly bad results, which is shown and discussed in chapter 4.5. In regular expressions tests, C++ and Perl were overwhelmingly good in execution times, and they also had lowest memory usages. Overall, Perl had the best results for maximum RSS usage.

In maximum RSS usage, it should be noted that the values included the amount of memory the language required to load the string (.txt-file). For example, on strings sizes of 100 million characters, Java used 753 MB memory on loading the file, and Swift used 389 MB, which made them stand out in maximum RSS usage tests. Another option would have been to subtract the maximum RSS usage value for loading strings from the operation's maximum RSS usage values which include loading the strings. However, the problem in that option was that sometimes it leads to values smaller than zero, because maximum RSS values change in every run (see chapter 3.7 for more details). It isn't also necessarily wrong to include the values for maximum RSS usage on the whole process, because it simply shows how much memory was really used on the operation. In execution times, it wasn't possible to have values smaller than zero, because execution times were measured inside programs using system clock. Nevertheless, execution times for loading only the string files were done for comparison. Chapter 4.1 will discuss execution times and memory usages for loading the string files more in detail.

This chapter will show the results as R plots. From now on, string sizes will be abbreviated: e.g. string size of 10 million characters will be written as "10m string" ("m" standing for millions, and "k" standing for thousands). See Appendix A for more exact values.

4.1 Load string file

This chapter presents the results for loading string files only. Figure 1 shows the execution times and Figure 2 shows the maximum RSS usages for loading the string files. In other results between chapters 4.2 and 4.14 it should be noted that values for maximum RSS usages include loading string files, but in execution times the values are only for the operation, and they don't contain the times for loading string files.

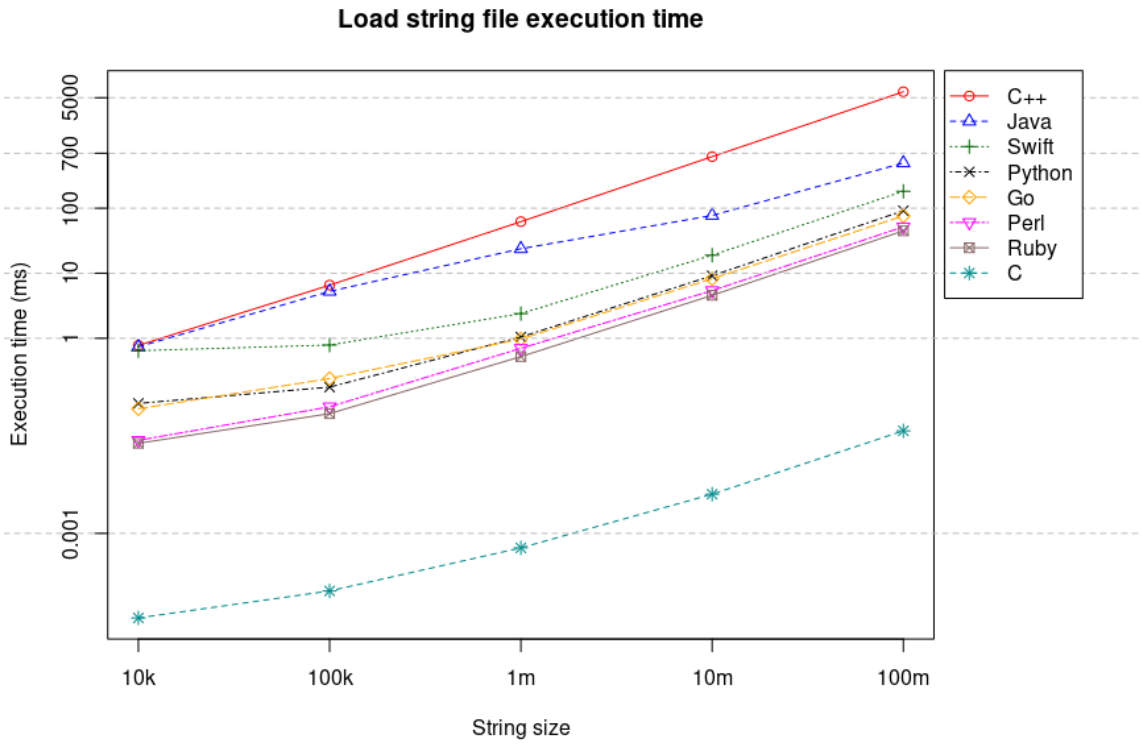

Figure 1. Execution times for loading string files.

C++ was by far the slowest language on this task with execution time of about 6.2 seconds 100m string, second being Java with 0.5 seconds. C was clearly the fastest with 0.04 milliseconds.

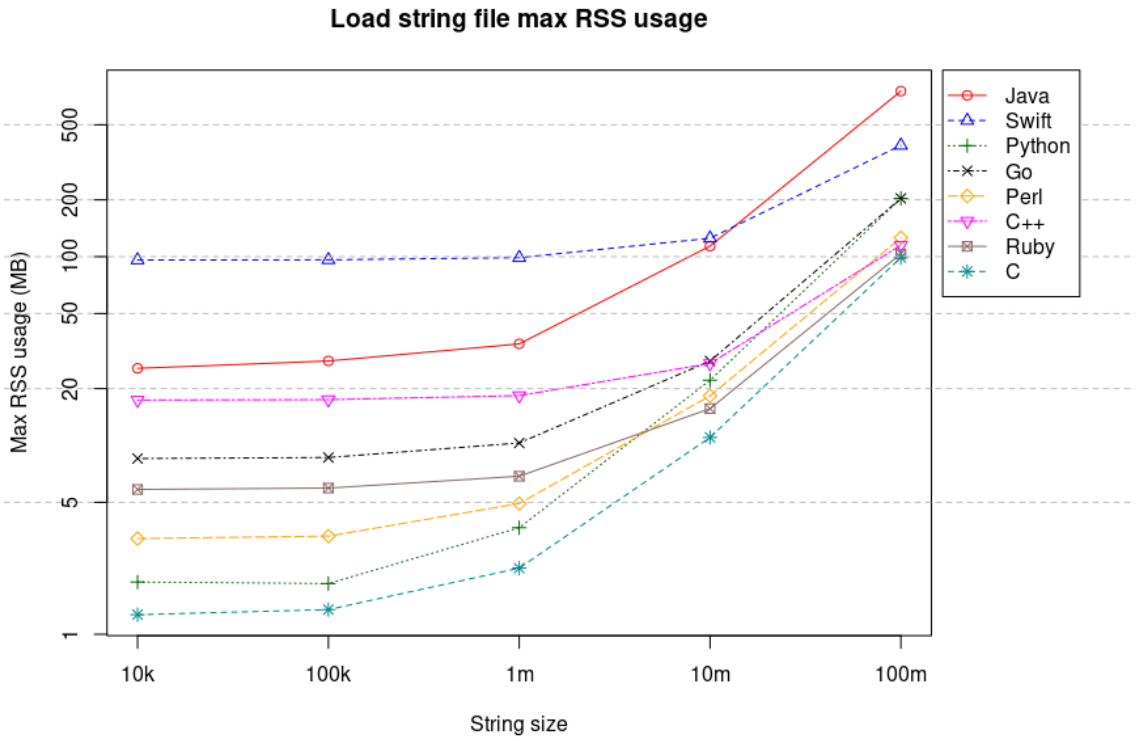

Figure 2. Maximum RSS usages for loading string files.

Memory usages are shown in Figure 2. Swift used almost 100 MB memory in loading the smallest string size, and Java used the most memory in loading the largest string size.

4.2 Concatenation

In string concatenation, each program had to simply concatenate character “a” into the string. The operation was simple, but the results were varying. Figure 3 shows the execution times, and Figure 4 shows the maximum RSS usages on the operation.

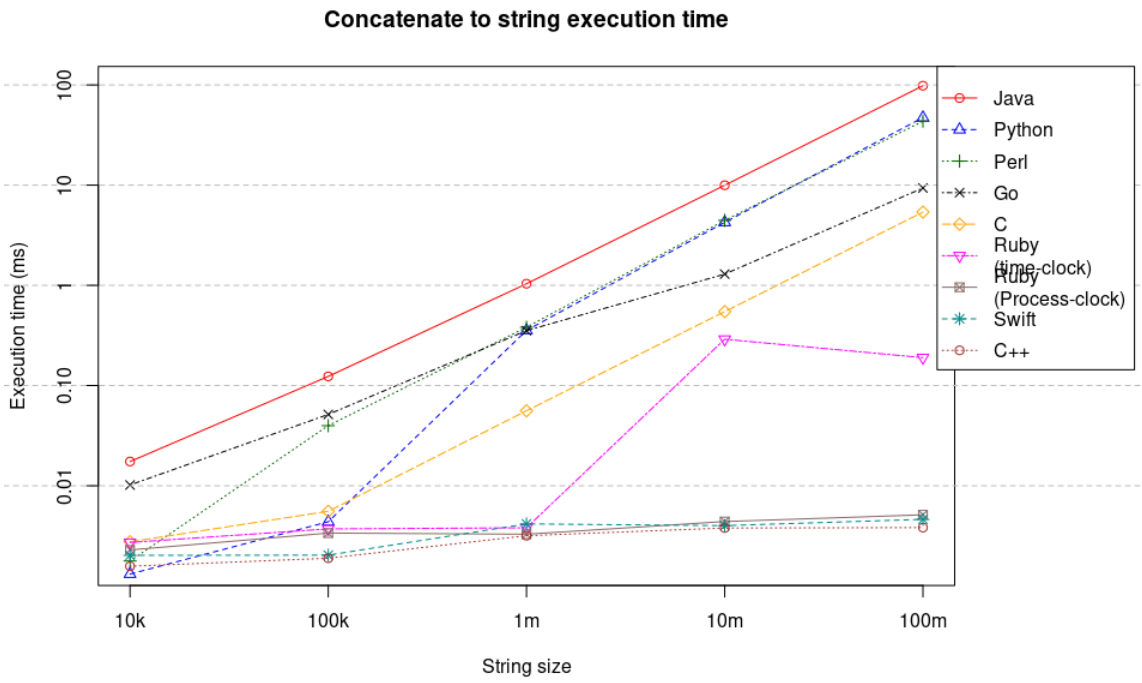

Figure 3. Execution times for string concatenation.

On 100m strings, the highest execution times were for Java (98.2ms), Python (47.2ms) and Perl (43.4ms). C++ (0.004ms) and Swift (0.005ms) had the best results in execution times. The increase from the fastest times to Java's 98.2ms was a massive 2,454,900%. Overall, compiled languages had better results for execution time than scripting languages or Java, as four out of five fastest languages were compiled languages.

Execution time for concatenating to string in Ruby had a somewhat strange value for 100m string (values for both 10m string and 100m string were triple checked). It was noticed that these values occurred only when Ruby's Time-library was used for getting the execution time:

```
start = Time.now
```

However, the results were very different, when Process-library was used instead:

```
start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
```

To illustrate the differences, both of the values were added to the graph in Figure 3. It remains unclear what caused such peculiar values for execution times when Time-library was used. Appendix B shows how the values changed across different string sizes when Time-library was used. In addition, concatenation was the only operation in which such values occurred; this was tested by running all operations with Process-library, and the results were the same.

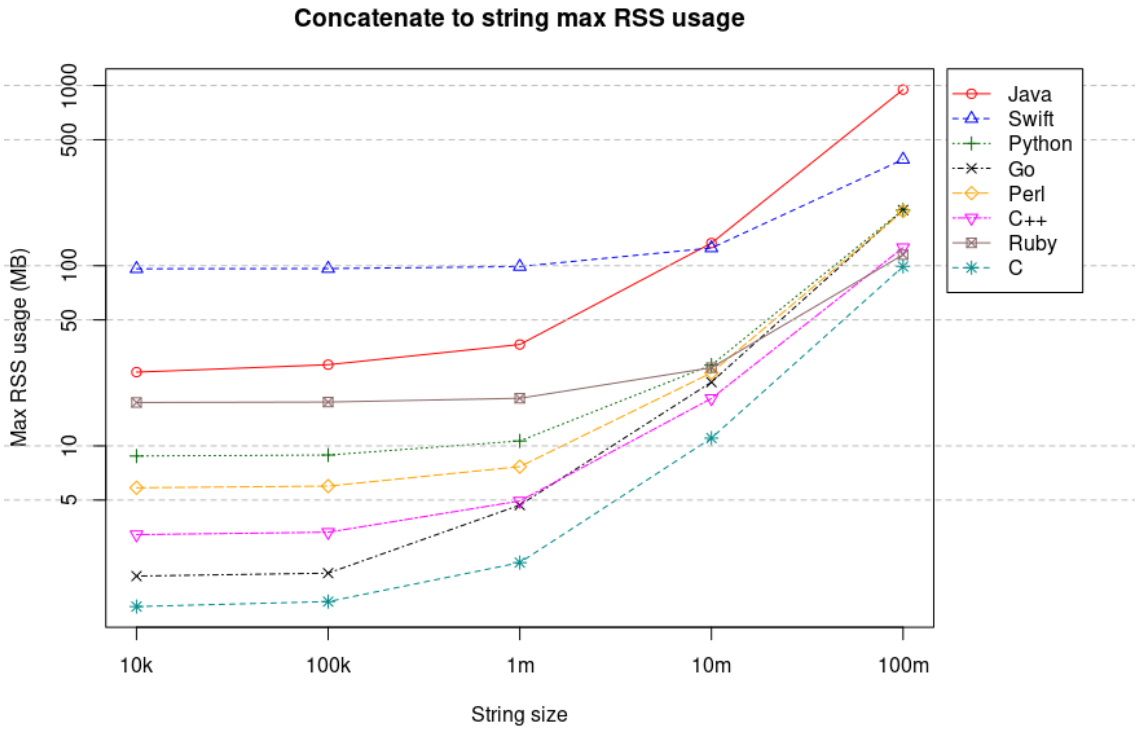

Figure 4. Maximum RSS usages for string concatenation.

Figure 4 shows the maximum RSS usages for string concatenation. Java had the highest memory usage (949 MB), and the increase from loading the string file (753 MB) was 26%, or 196 MB. Perl's increase from 104 MB to 201 MB was 93.3%, or 97 MB. Other languages (Swift, Python, Go, C++, Ruby, and C) didn't have any increase in memory usages. Strings in Java, Python, Perl, and Go are immutable, which could be the cause for those four being the four slowest languages in this test, and for Java and Perl having a peak in memory usage.

4.3 Replace

In replace operation, the programs were to replace all substrings of "abba" with "CCCC". The execution times are shown in Figure 5 and maximum RSS usages are shown in Figure 6. C, C++ and Perl didn't have any built-in function for replacing substrings.

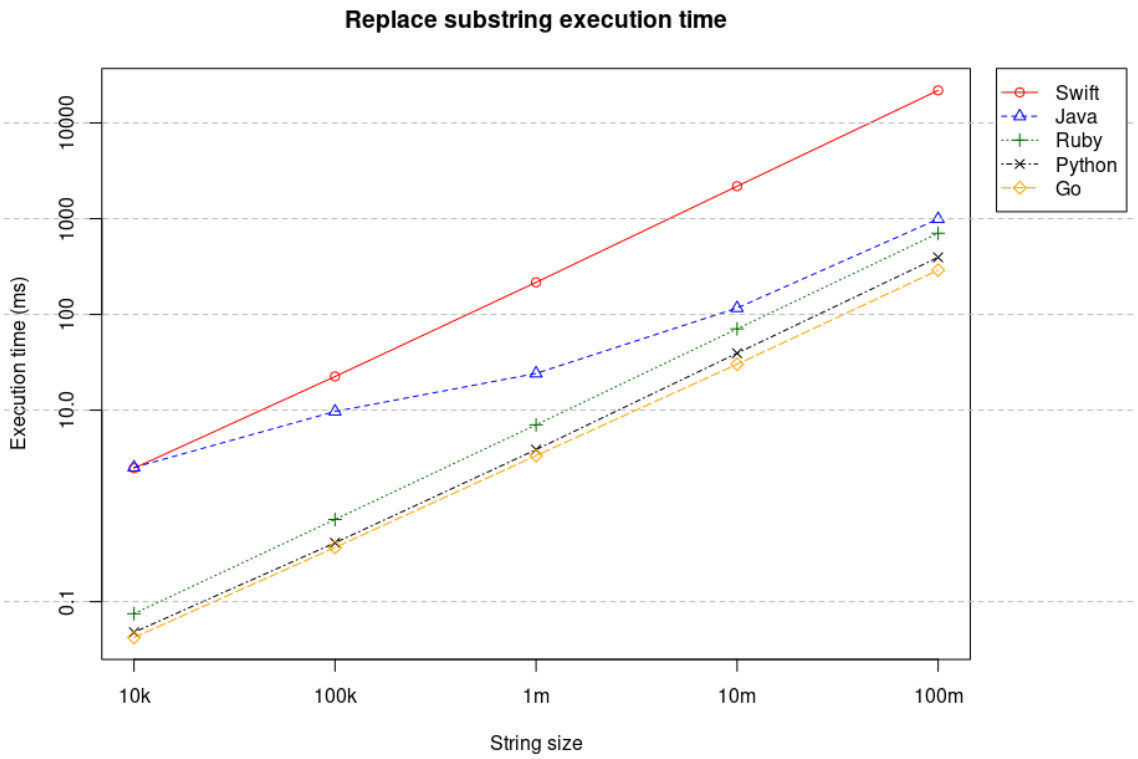

Figure 5. Execution times for replacing substrings.

Swift had clearly the worst execution time for 100m string: 21,9 seconds. Java was the second worst with 1 second (making a huge gap to Swift), Ruby was third (0.7s), Python was the second fastest (0.39s) and Go was the fastest (0.29s). The increase from Go's execution time to Swift's was 7,440%.

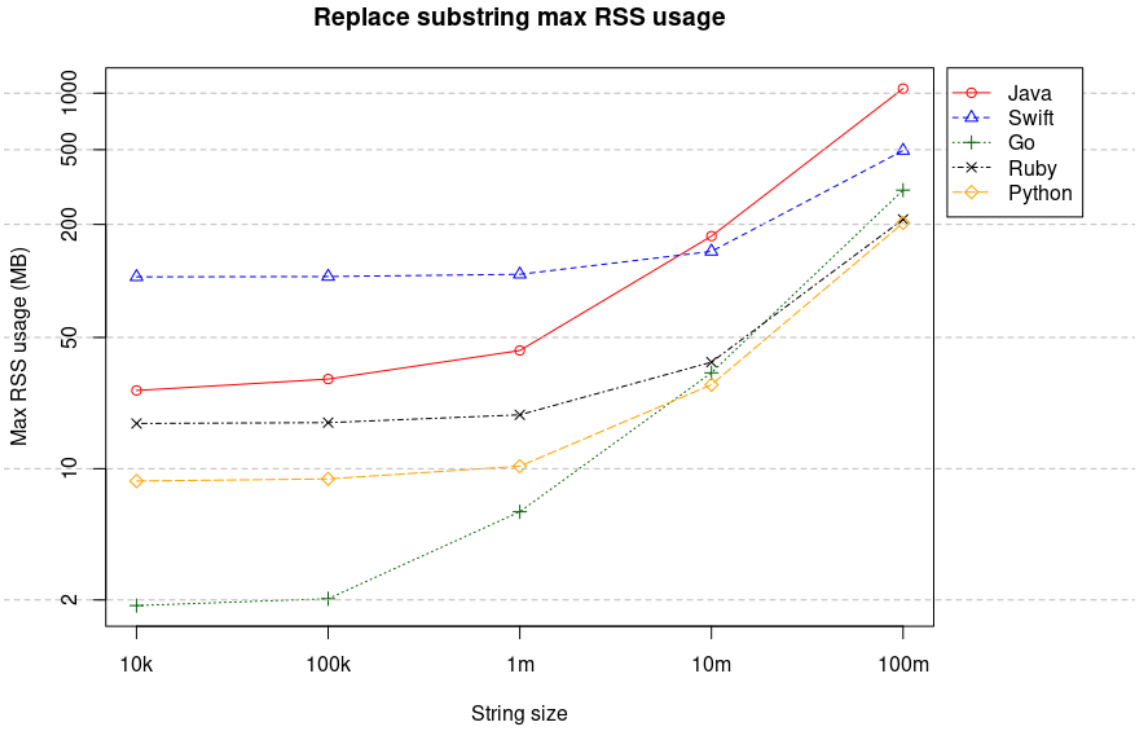

Figure 6. Maximum RSS usages for replacing substrings.

As for memory usages, Java's maximum RSS usage was 1,058 MB (increased by 41%/305 MB) and Swift's was 495 MB (increased by 27%/106 MB), and these two used the most memory. Go used 304 MB (increased by 49%/100 MB), and Ruby's usage was 213 MB (increased by 85%/98 MB). Python didn't use any additional memory compared to loading the string.

4.4 Reverse

In reverse operation, the programs had to, as the name suggests, convert a string into reverse order. The results are shown in Figure 7 (execution times) and Figure 8 (maximum RSS usage). C has a built-in function "strrev" for reversing a string, but it is not available for Linux-version of gcc, so it had to be excluded, and Go didn't have a built-in function for the operation. Perl's reverse-function works only with scalars, so the string had to be converted into scalar first. Similarly in Java, the operation was carried out by first creating a new StringBuilder-type from the string, which was then reversed and converted back to a String-type. Also, it should be noted that Python doesn't have an actual built-in method for reversing a string, but the following line was used to reverse a string:

```
str[::-1]
```

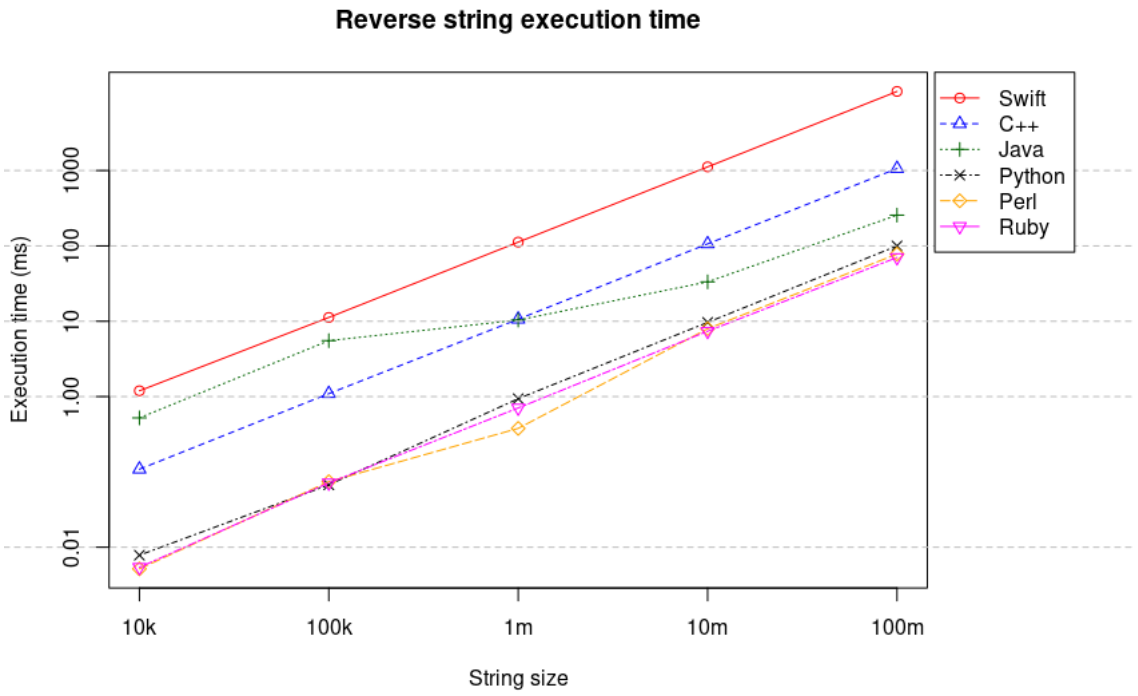

Figure 7. Execution times for reversing a string.

Swift was the slowest language on the operation on biggest strings, as it took 11.2 seconds on average to run the operation on 100m string. The second slowest was C++, which executed in 1.1 seconds on 100m string, which is surprisingly high, considering that C++ is a fast language in general. The drop from Swift to C++ is also noteworthy; Swift was over ten times slower than C++. The execution time for Java was 0.26 seconds, and Ruby was the fastest with 0.07 seconds along with Perl with 0.08 seconds. As it can be seen from Figure 7, Java's performance relatively improved starting from 1m strings.

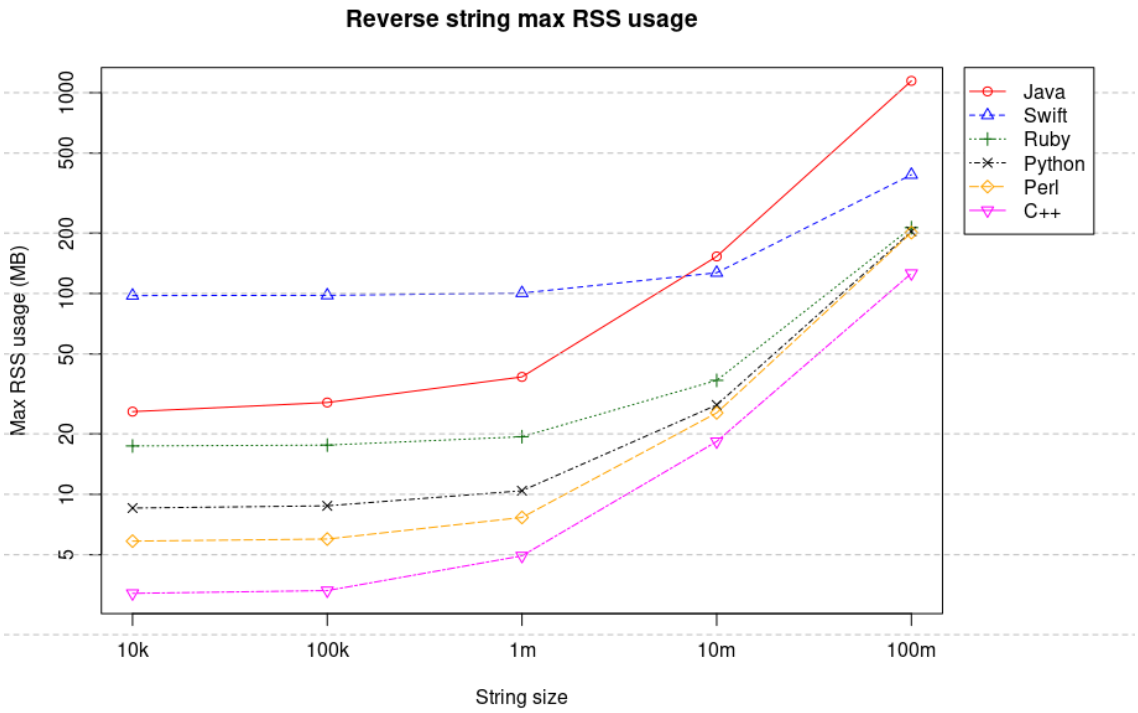

Figure 8. Maximum RSS usages for reversing a string.

The worst results for maximum RSS usage in reversing a 100m string were for Java with 1,146 MB memory used, meaning that the increase was 393 MB/52%, from loading 100m string file. Ruby's memory usage increased from 115 MB to 213 MB (increased by 85%), and Perl's memory usage also increased by 97 MB (93%). Swift, C++, and Python didn't have any increase in their memory usages compared to loading strings, which is a bit surprising for Swift, because it was clearly the slowest language in this operation.

In both tests, scripting languages (Python, Perl, Ruby) had very similar results. In execution times, scripting languages had the best results, and in memory usage scripting languages were only beaten by C++. Thus, it can be concluded that scripting languages outdo compiled languages in reversing operation.

4.5 Sort

In sort operation, the program had to sort a string. It wasn't defined whether the program had to sort strings into ascending or descending order, and every language sorted strings into ascending order by default. Figures 9 and 10 show execution times and maximum RSS usages for sorting a string. The standard libraries of Java, and Perl don't have a sorting function. C has qsort-function, but implementing it requires using a compare function, so it was decided to rule it out.

Ruby's sort doesn't work with ordinary strings, and first the string had to be splitted before sort would work, and after sorting it had to be joined again. In Go, the string had to be splitted, too, and Go's sort converts characters into a string array. Swift's sorted-method and Python's sorted-function also first convert the string into a string array.

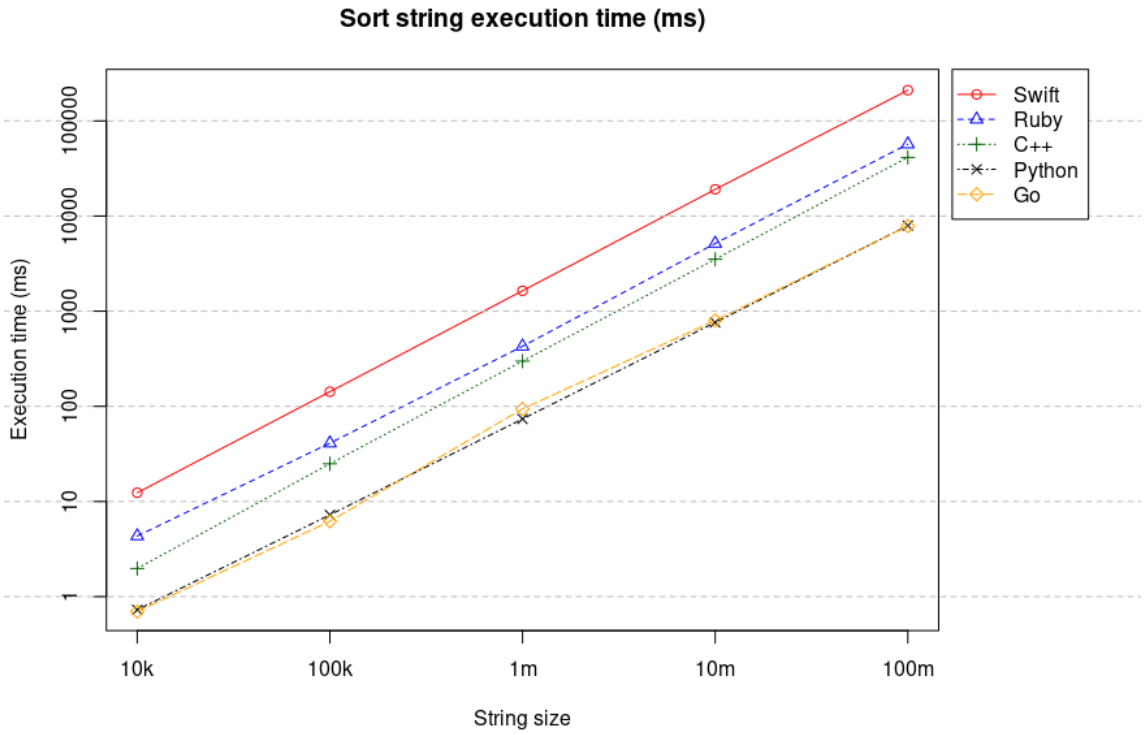

Figure 9. Execution times for sorting a string.

Swift was clearly the slowest language, as it had 3-4 times higher execution times in all string sizes than the second, Ruby. Sorting a 100m string took 210.4 seconds for Swift, 57 seconds for Ruby, 41.4 seconds for C++, 7.9 seconds for Go, and 7.9 seconds for Python (Go was slightly faster than Python). Go was the fastest language, although it used the second most memory. C++ was in the middle, being outdone by Python and Go. The increase from Go's execution time to Swift's time on 100m string was 2,581.7%.

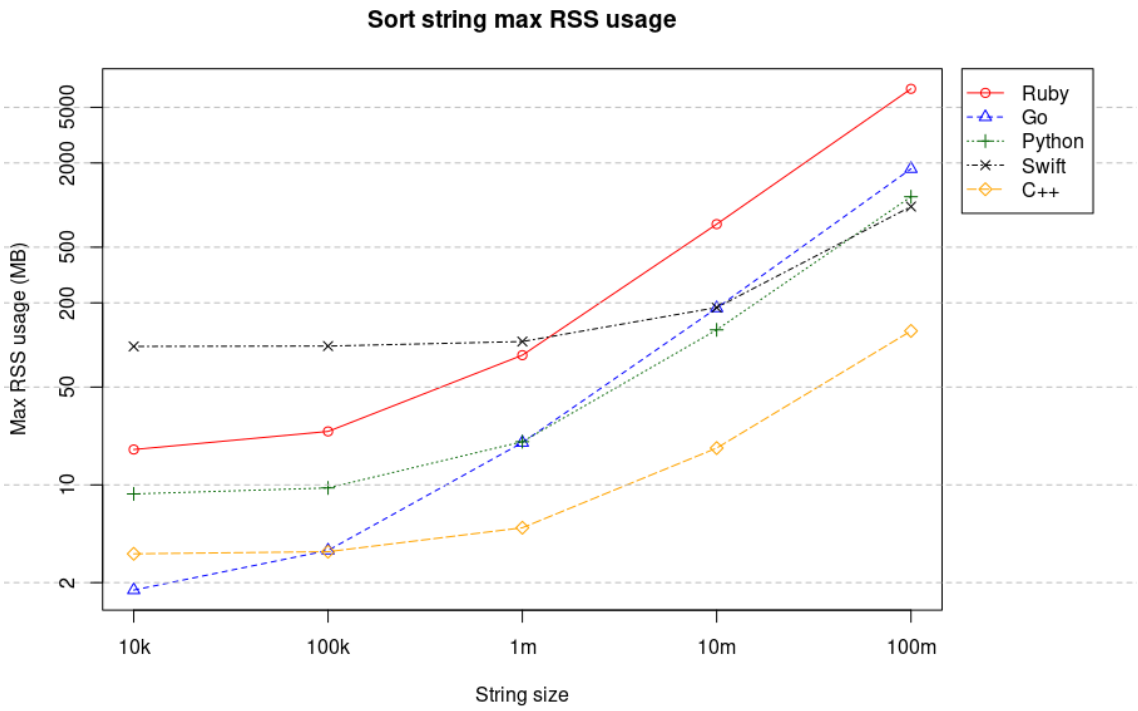

Figure 10. Maximum RSS usages for sorting a string.

Although the results clearly favored Python and Go in execution time, that wasn't quite the case in maximum RSS usage, as Go had the second worst results for 100m string with 1,817 MB memory used (increased by 791%/1,614 MB), and Python was in the middle with 1,147 MB used (increased by 462%/943 MB). Ruby had clearly the worst results for memory usage, as it used a whopping 6,776 MB (increased by 5,792%/6,661 MB). Although Swift was clearly the slowest language in this operation, it had the second best results for maximum RSS usage on 100m string with 976 MB used (increased by 151%/587 MB). C++ had the best results for memory usage as it didn't use any more memory compared to loading the string file.

4.6 String duplication

In string duplication, the program had to duplicate the string that was read. Ruby, Python, and C had a built-in method or function for duplicating a string, but Perl, C++, Go, and Swift didn't, so string duplication was accomplished by a simple variable assignment on those languages. In Java this was accomplished by creating a new string using new-operator. Execution times in string duplication are shown in Figure 11, and maximum RSS usages are shown in Figure 12.

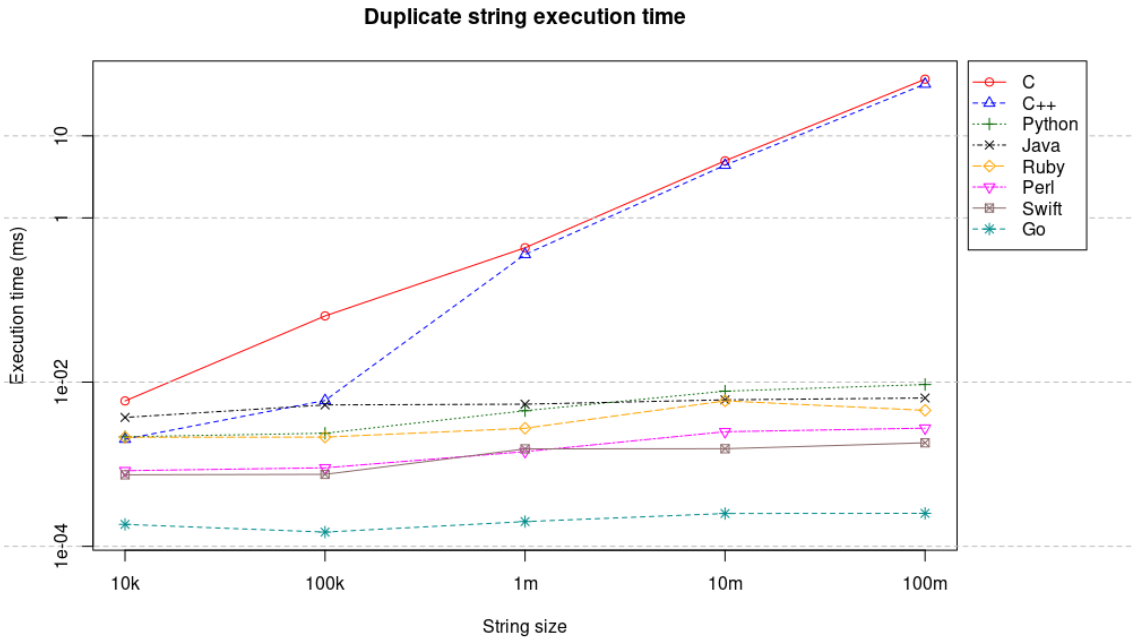

Figure 11. Execution times for string duplication.

The highest execution times for string duplication were for C (49 milliseconds on 100m string) and C++ (43 milliseconds on 100m string) from 100k strings up to 100m strings. In C, strdup-function was used, and in C++ duplication was done by variable assignment. Other languages had really short execution times for the task, Go being the fastest with 0.0003 milliseconds on 100m string.

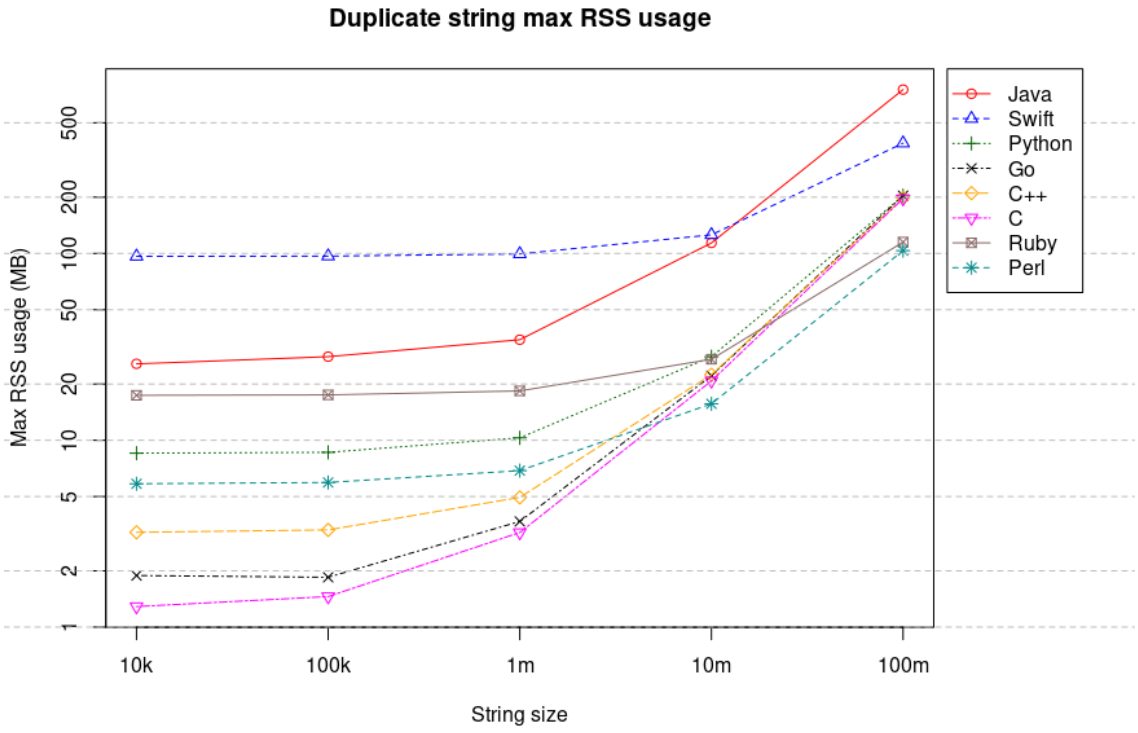

Figure 12. Maximum RSS usages for string duplication.

Despite C and C++ were clearly the slowest languages on string duplication, maximum RSS usage for C++ (198 MB, increased by 57%/72 MB) was the fourth least, and for C (197 MB, increased by 101%/98 MB) it was the third least. All other languages didn't use more memory than they did for loading string files. Thus, it's clear that C and C++ had the worst results for string duplication, as they were the slowest languages and they were the only languages that used considerably more memory on the operation compared to memory required for loading the string file.

4.7 Find first index of substring

The programs had to find the first index of a substring in this operation (e.g. C-function strstr). The substring that needed to be found was "abba". The graphs are presented in Figures 13 and 14. Swift didn't have any built-in method for the operation.

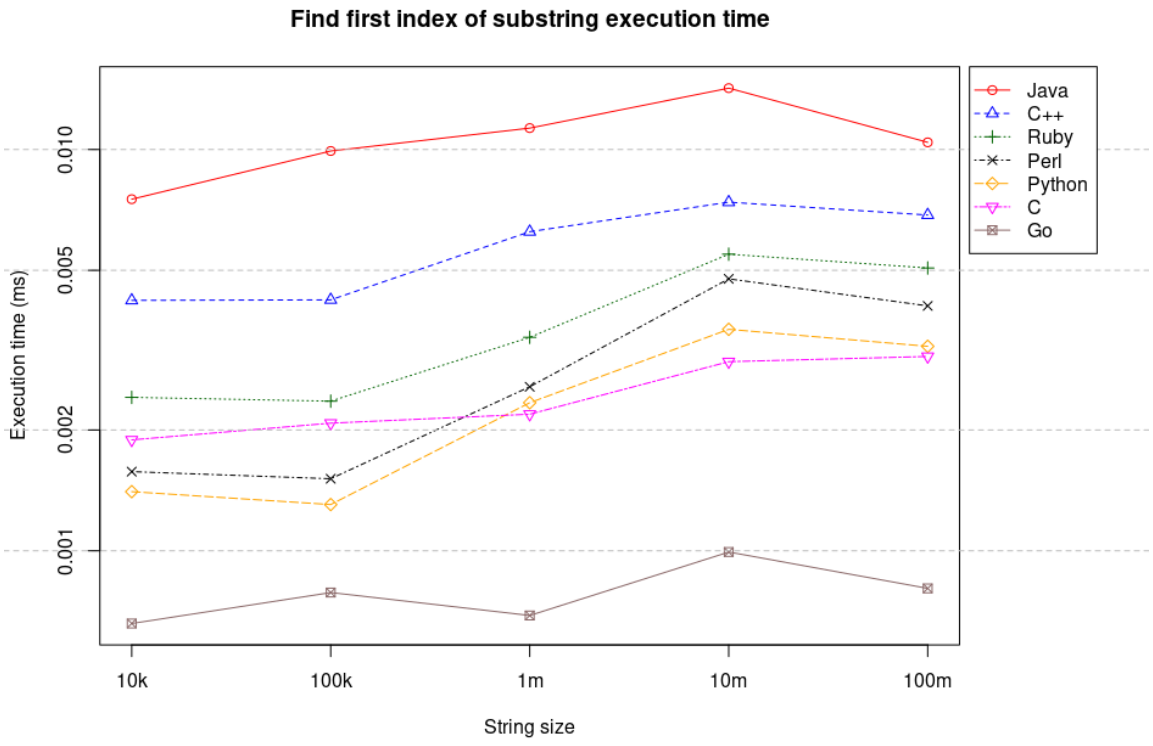

Figure 13. Execution times for finding the first substring in a string.

Finding the first index of a substring is very light operation in itself, and this can be seen from the execution times, too. It can also be seen that in 100m string there was a match at an early point of the string, because the execution times were decreased from 10m string values. Go was the fastest language with all under 0.001ms execution time on every string size, and Java was the slowest with execution times varying between 0.007ms-0.015ms.

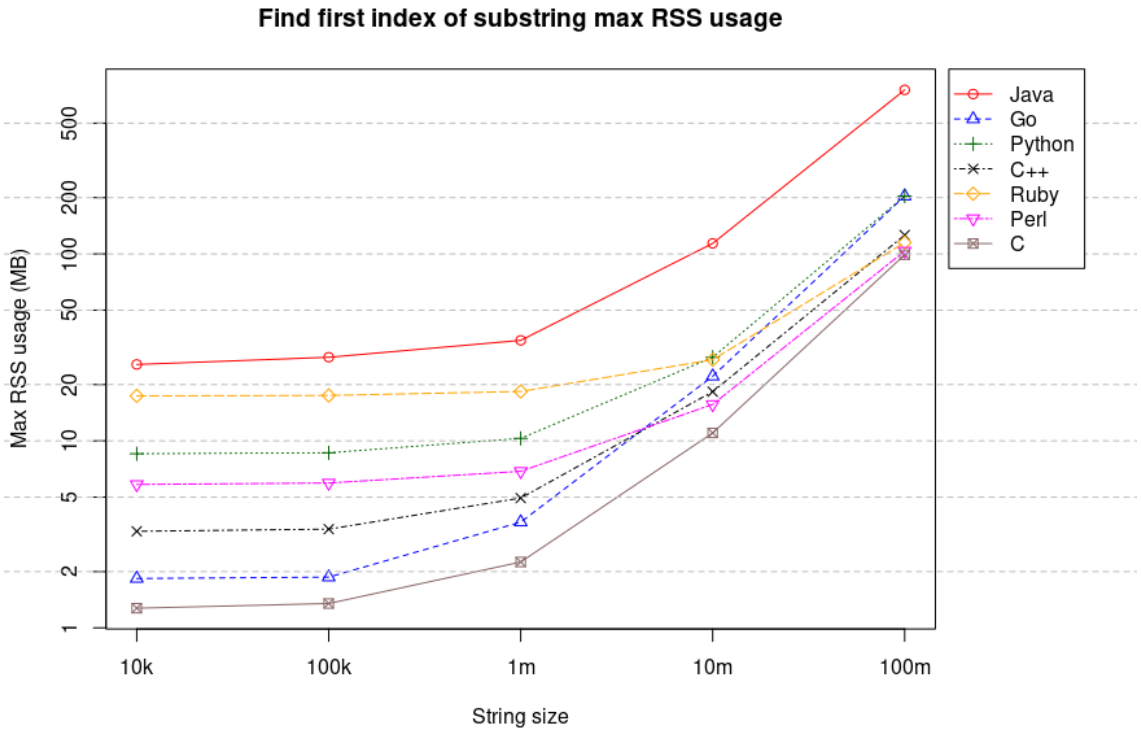

Figure 14. Maximum RSS usages for finding the first substring in a string.

There wasn't any increase in memory usages in any languages compared to loading string files, which is a finding in itself; based on both execution times and memory usages, it can be said that finding the first index of a substring is "the lightest" operation in this thesis. The reason for this is that the strings consisted of only three characters, so the first occurrence of substring "abba" will most likely be found faster than it would be if the strings would consist of, e.g., 20 different characters.

4.8 Uppercase

Uppercase operation means converting every character into an uppercase character. In C and C++, uppercase was implemented with a while-loop. Figure 15 shows execution times, and Figure 16 shows maximum RSS usages for the operation.

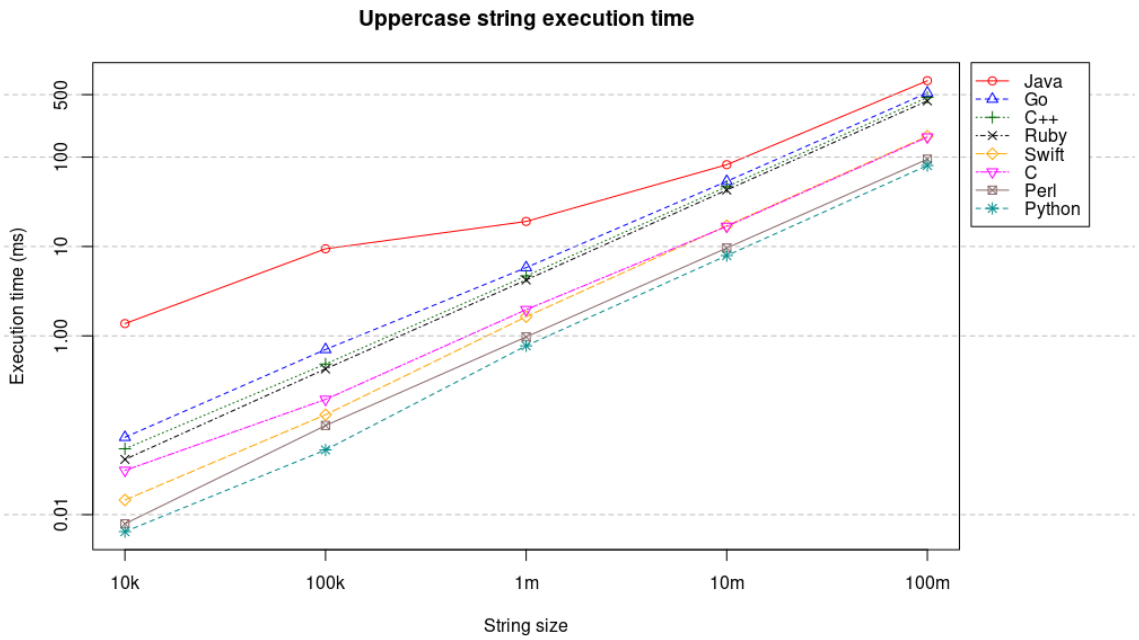

Figure 15. Execution times for converting strings to uppercase.

Java was the slowest language in uppercase string tests (716ms on 100m string), followed by Go (524ms). C++ (465ms) and Ruby (426ms) were pretty tied, and so were Swift (172ms) and C (168ms). Perl and Python were the fastest languages, and Python turned out to be slightly faster than Perl: on 100m string, Perl's execution time was 95ms, and Python's was 81ms. The increase from Python's execution time to Java's time on 100m string was 789%.

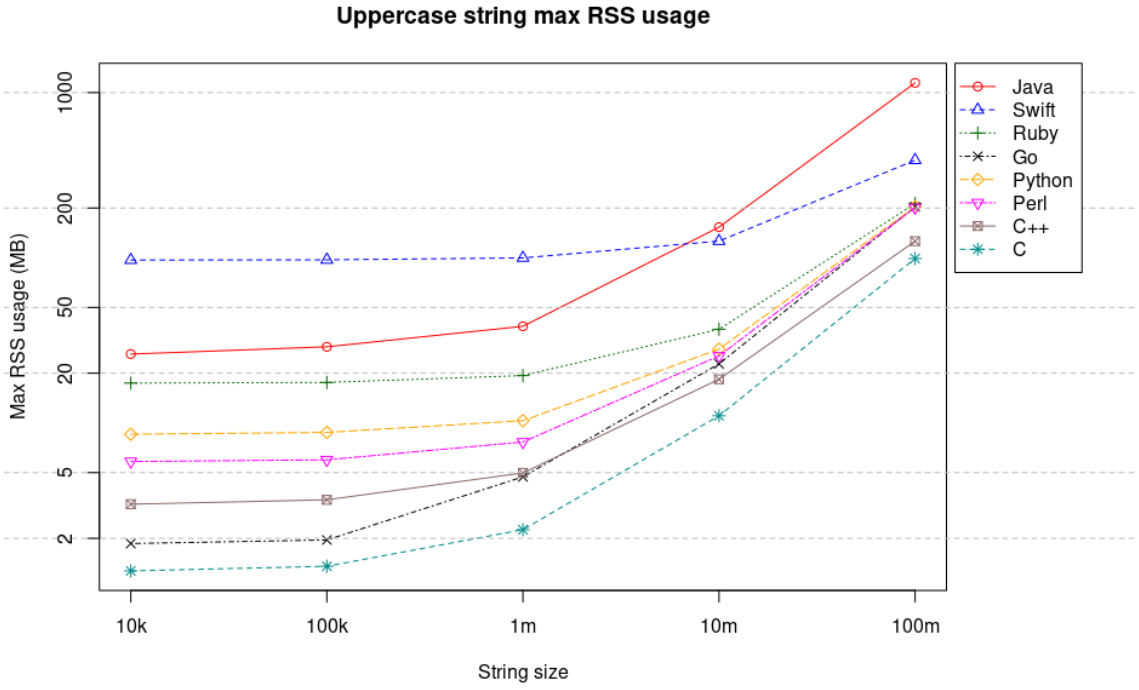

Figure 16. Maximum RSS usages for converting strings to uppercase.

Java had the highest maximum RSS usage: in 100m string, Java's memory usage increased to 1,146 MB from 754 MB (percentual growth of 52%) that was used in loading string file. Memory usage increased in Ruby from 115 MB to 213 MB, and in Perl from 104 MB to 201 MB. Other languages (Go, Swift, Python, C++, and C) didn't have any increase in memory usage. The increase in Perl and Java could be derived from strings being immutable.

4.9 String equality

Testing for string equality required naturally more work, because the string that the comparison was done against needed to be long enough for the test to be reasonable; for example, testing whether a string sized 10,000 characters is equal to a string with one character is not practical, because the test is too easy. Rather, the comparison string should be almost the same as the original string. To achieve this, a copy was made from the original string, and character "a" was appended to the end of the string. These strings were compared either with if-else-statement or with a built-in method or function. In this kind of setting, some languages might test string equality real fast in case the language tests first whether the strings compared are equally long. Therefore, more reasonable test would have been to copy the original string and change the last character only, but this would have been quite tough to achieve in some languages without having too much impact on memory usages.

An example of the equality test in Python:

```
str2 = str+"a"
if str is str2:
      pass
else:
      pass
```

Figure 17 shows execution times, and Figure 18 shows maximum RSS usages for the operation. In the following analysis, the results for string duplication and string concatenation will be used to analyse the memory usages of equality comparisons. See chapter 4.5 for string duplication and chapter 4.2 for string concatenation results, or Appendix A for more exact results.

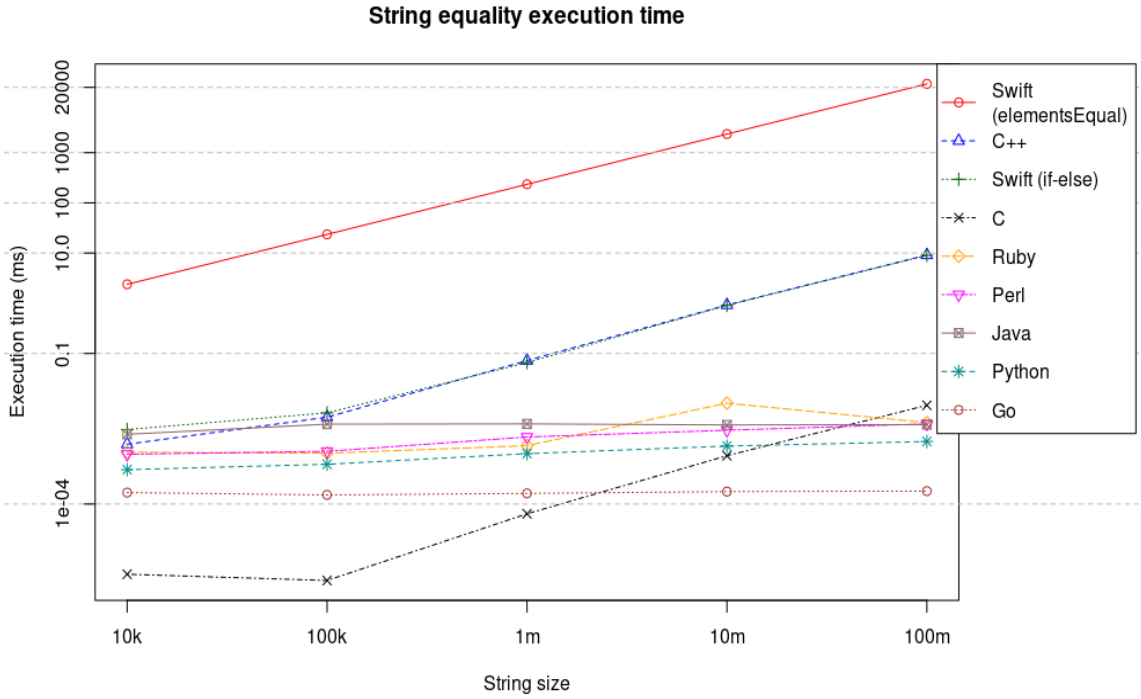

Figure 17. Execution times for string equality.

Swift fell far behind all other languages in execution time (23.5 seconds on 100m string), as the second most time took for C++ (9ms). Based on the source code of the Swift's elementsEqual-method, it seems that the method is purposefully implemented to be a generic method that could be used to compare various different types of elements, which naturally lowers performance. It also uses iterators, which should use some time. I tried to find out what possibly was causing such high execution time by first copying the method from the source code, and then by modifying the method, but I wasn't able to get any remarkable results. String assignment with =-operator was accomplished for 100m string in 0.002ms in string duplication operation (see chapter 4.5), so that isn't the case. Also, when Swift's elementsEqual-method was replaced with ==-operator and an if-else-statement, the execution time for 100m string was 9ms (decreased by 99.96%). It could be that the ==-operator either loops through the strings to find out whether they match, which isn't time consuming, or that the operator actually first compares whether the string-objects match. Regardless of the decrease in execution time when if-else statement was used, Swift was still the second slowest language - C++ was only marginally slower.

All other languages executed so fast that there is no need to analyse them, because the variance of execution times have so big an impact on such small values (in other words, the difference between 0.009ms and 0.006ms might be derived from a background process).

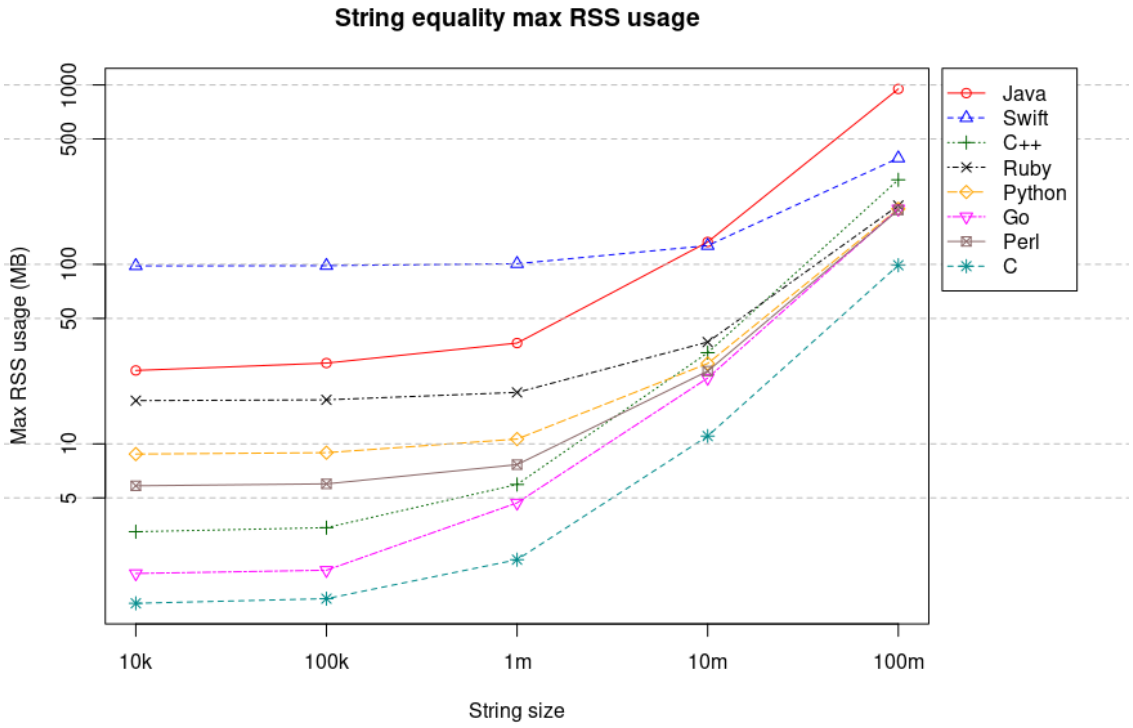

Figure 18. Maximum RSS usages for string equality.

Java had the highest maximum RSS usage with 950 MB, and the increase of 197 MB was derived from string duplication. C++ used 296 MB memory (increased 171 MB), which is the highest value for C++. Upon closer look, string duplication used 198.4 MB memory, but concatenation didn't use more memory than loading the string file, so the compare-function of C++ used about 98 MB memory. Ruby used 213 MB (increased by 98 MB from loading the string file), and what is interesting in the increase is that neither concatenation to string nor string duplication increased memory usage in their tests, but when these two were combined, memory usage increased by almost 100 MB. However, because it was only those two operations combined that increased the memory usages, it can be concluded that the equality comparison itself didn't increase the memory usage. Perl's memory usage was 201 MB (increased by 97 MB), caused by string concatenation. C's tests were implemented using a combination of strdup-, strcat- and strcmp-functions, and C used around 197 MB memory, which is caused by strdup-function used, so strcmp-function used for equality comparison didn't use any additional memory. Memory usages for Swift (on both elementsEqual and if-else statement), Python, and Go didn't have any increase from loading the string file. This is a bit surprising for Swift considering how overwhelmingly slow elementsEqual-method was. In a nutshell, the only language that used any additional memory for the actual equality comparison was C++.

4.10 Regular expressions

This chapter consists of five sub-chapters, each for one regular expression test. The operations are explained more in detail in their respective chapters. From now on, regular expressions are abbreviated as regex.

Tests for regular expressions show that Perl and C++ were in a completely different class than other languages, at least for execution times. Both in execution times and

maximum RSS usages Perl had the best results in every regex test, and C++ was the second, but Perl was only slightly better than C++. In memory usages the differences weren't as big as they were in execution times. It seems that both C++ and Perl are highly optimized in regex operations, because their execution times don't increase when string sizes grow. Appendix C shows the summaries of all tests by positions, although it doesn't show the difference between values across languages.

The languages that didn't use any additional memory in regex tests were Python, C++, Java, and Perl. In other words, Ruby, Swift and Go were the only languages in which memory usage increased from loading the string file, but the increases weren't that high: in Ruby, the growth in memory usage was 19-47 MB, in Go it was 36-63 MB, and in Swift the growth was 50-88 MB. In addition, in every regex operation the positions across languages were the same; in other words, in every operation, Perl used the least memory, Java used the most memory, and so on. The fact that memory usages didn't increase by much could indicate that regular expressions are generally lightweight operations in terms of memory usage; in other operations, there was usually some language that consumed significant amounts of memory compared to loading string files only, but none of the regular expressions tests used that much memory. However, it should be noted that the regular expressions tested in this thesis were pretty simple, because the strings contained only three different characters. The memory usages for regular expressions won't be analyzed/documented operation-wise. See Appendix A for exact values.

Overall, Swift was the slowest language on regex tests: in three tests out of five, it was the slowest, and in two tests it was the second slowest. Go was fast in regex 1 and 2, but for the rest it was really slow, which is an interesting finding in itself. Ruby had quite decent execution time results overall, though it was nowhere near C++'s and Perl's results. Python placed somewhere in the middle with no exceptionally great or bad execution times.

Regular expressions for Java were implemented by using a combination of Pattern, Matcher, and a while-expression. In Swift, NSRegularExpression-expressions were used. Go's regular expressions used MustCompile-function from regexp-library along with FindAllString-function. All implementations are available at GitHub (see Appendix E).

Regular expressions weren't implemented in C, because C's standard library doesn't have a function for regular expressions, and an external library must have been used.

4.10.1 Regex 1

Regex 1 was about finding all substrings that contain "ba" followed by exactly three "b" characters. Figure 19 shows execution times for the operation. An example of regular expression 1 in Python:

```
re.findall(“bab{3}”, str)
```

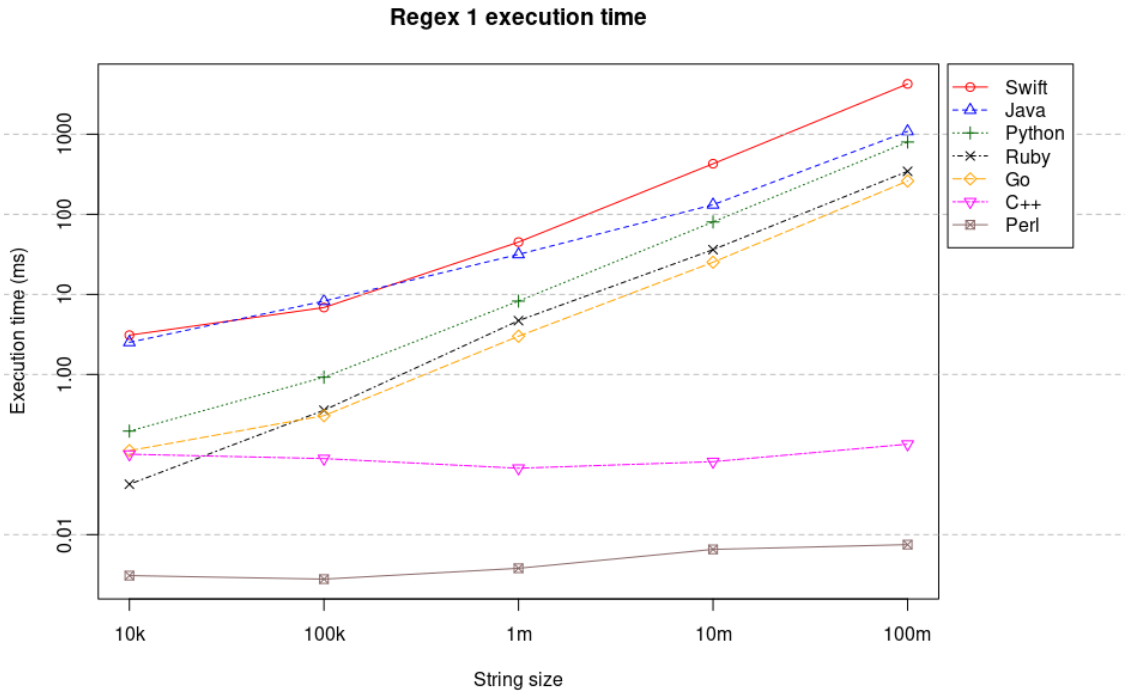

Figure 19. Execution times for regex 1.

Swift executed in 4.2 seconds on 100m string, being the slowest language on the operation. Java was the second slowest (1.1s), followed by Python (0.8s) and Ruby was in the middle (0.34s). Go was fastest after C++ and Perl with execution time of 0.26 seconds on average.

4.10.2 Regex 2

In regex 2, the programs had to find all substrings that contain “bbbb” followed by one or more “a” characters. Execution times for regex 2 are shown in Figure 20. An example of regex 2 in Python:

```
re.findall(“bbbba+”, str)
```

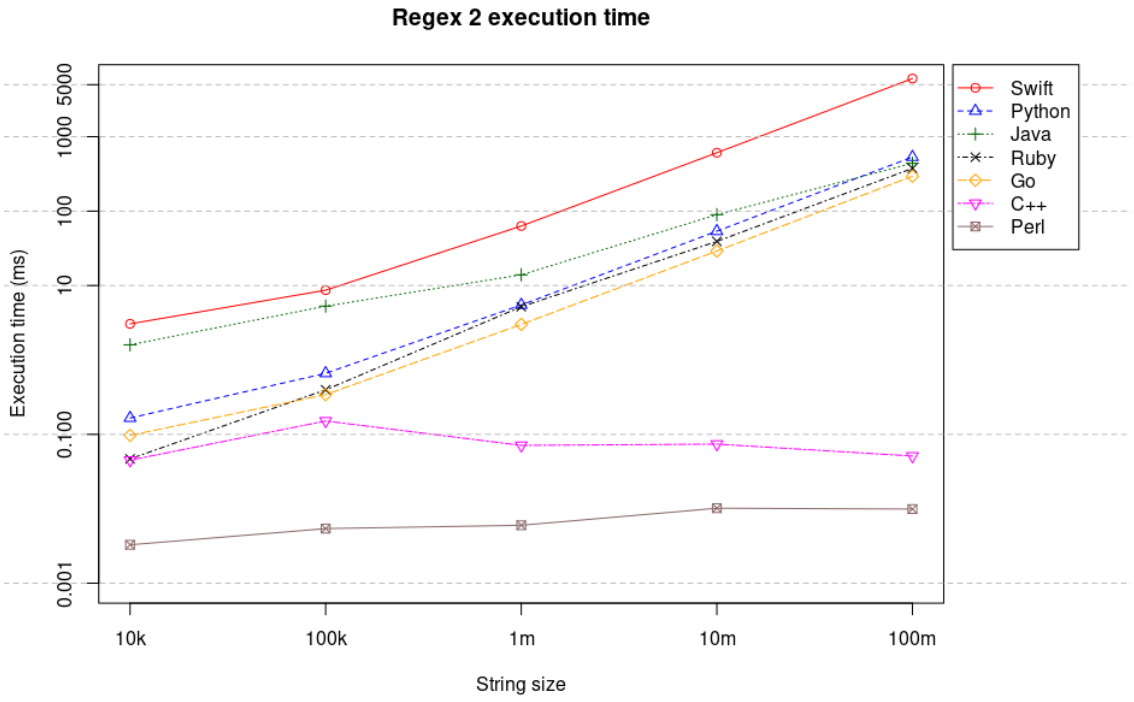

Figure 20. Execution times for regex 2.

The slowest language was Swift (4.1 seconds on 100m string). Python was the second slowest language on 100m string (0.53 seconds), but it was considerably faster than Java up to 10m strings. C++ and Perl excluded, Go was the fastest language among the five others (0.29s on 100m string).

4.10.3 Regex 3

In regex 3, the programs had to find all substrings that contain either “bbbbb” or “aaaaa”. Execution times for regex 3 are shown in Figure 21. An example of regex 3 in Python:

```
re.findall(“bbbbb|aaaaa”, str)
```

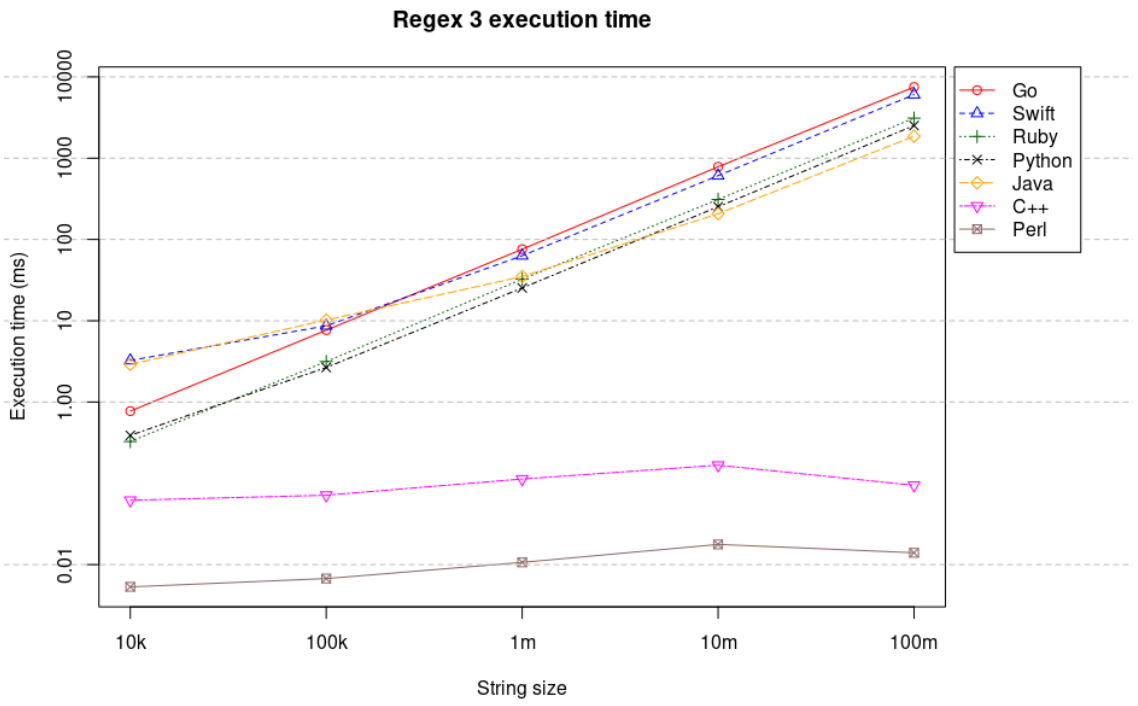

Figure 21. Execution times for regex 3.

This was quite demanding operation in terms of execution speed. As in every regex test, C++ (0.09ms) and Perl (0.01ms) were tremendously fast, and the fastest after them was Java, in which execution time was 1.9 seconds on 100m string. The execution times that followed after Java on 100m string were 2.5 seconds in Python, 3.1 seconds in Ruby, 6 seconds in Swift, and 7.5 seconds in Go. Java was relatively slow up to 1m string, but on 10m and 100m strings its' execution times improved.

4.10.4 Regex 4

Regex 4 is about finding all substrings that contain either "ba" followed by exactly three "b" characters or "bbb" followed by one or more "a" characters. Figure 22 contains execution times for regex 4. An example of regex 4 in Python:

```
re.findall("bab{3} | bbbba+", str)
```

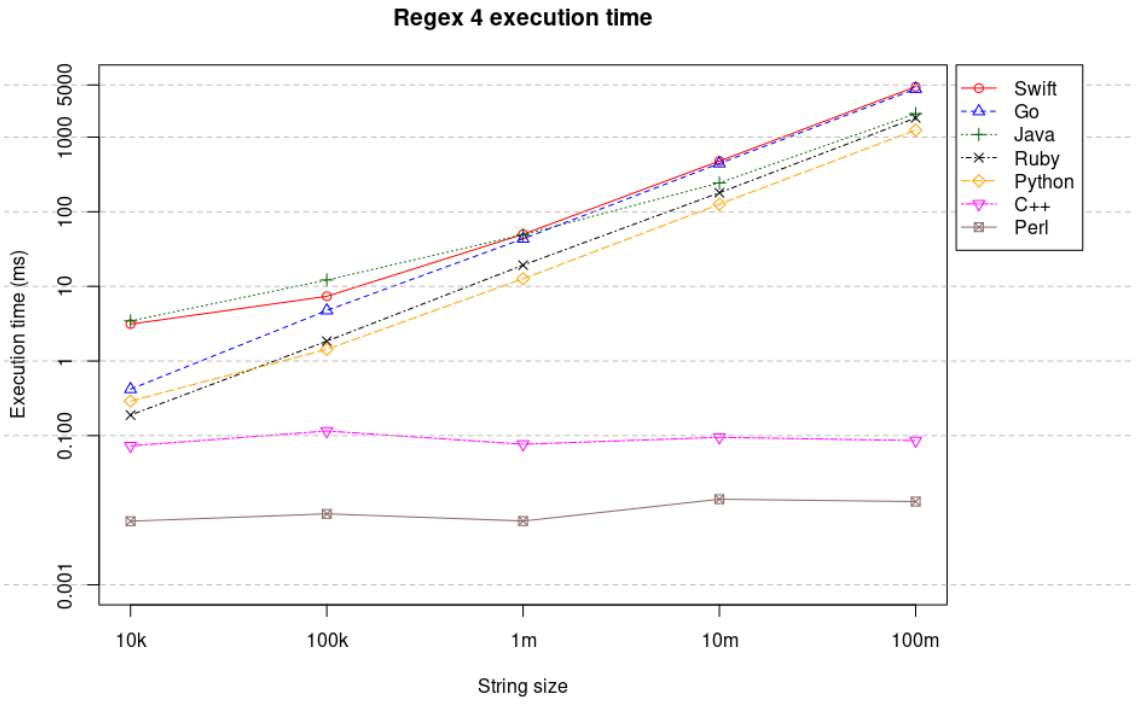

Figure 22. Execution times for regex 4.

The slowest languages were pretty even: on 100m string, Swift executed in 4.7 seconds and Go in 4.4 seconds. Java's execution time for 100m string was 2.1 seconds, but although Java was over twice as fast as Swift and Go on 100m strings, it was the slowest language up to 1m string. Ruby executed in 1.8 seconds and in Python it took 1.2 seconds.

4.10.5 Regex 5

In regex 5, the programs had to find all substrings that contain either "bbbbb", "aaaaa", or those five letters more than once in a row (e.g. ten "b" characters, 15 "a" characters, etc.). Figure 23 shows the graph for execution times. An example of regex 5 in Python:

```
re.findall("(bbbbb|aaaaa)+", str)
```

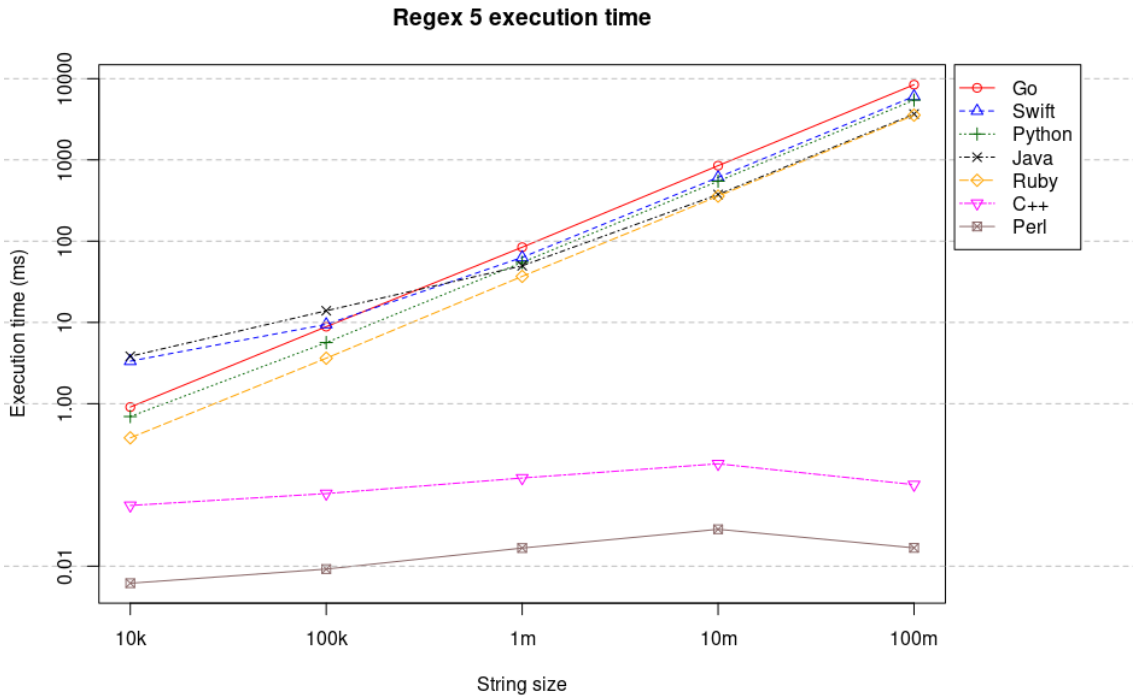

Figure 23. Execution times for regex 5.

Based on execution times, regex 5 was the heaviest operation: the fastest languages after Perl and C++ were Ruby with 3.6 seconds on 100m string, and Java with 3.7 seconds. The operation was executed on 100m string in Python in 5.4 seconds, in Swift in 6.9 seconds, and Go was the slowest language with 8.4 seconds. What is interesting is that in Java's execution times regex 4 and regex 5 had the same results up to 1m string, but on 10m and 100m strings regex 5 took almost twice as long to execute: times on 100m string were 2.1 seconds for regex 4, and 3.7 seconds for regex 5.

5. Discussion

There were some contradictory values on small string sizes, e.g. executing a program is faster on bigger string size than on smaller one, or that executing a program uses less memory than only loading the string file did. These results happened only between 10k and 100k strings, and they are derived from the background operations of the operating systems. However, these situations happened rarely, and they don't endanger the results of this thesis; despite there were some irregularities, it doesn't change the fact that they happen on every language, and the irregularities are only tiny ones that don't change the big picture.

Test strings used (strings with only a, b, and whitespace characters) are not quite like the most common examples would be where large string sizes emerge (e.g. string loaded from an Excel-file, or a DOM of a website). However, it was decided that it's better to use the test strings with only a, b, and whitespace characters, because they suit better for testing purposes (see chapter 3.2.).

It is widely believed that C and C++ are one of the most effective languages in terms of execution time and memory usage. This was also supported by Prechelt (2000), Fourment and Gillings (2008) and Pereira et al. (2017). In a study by Pereira et al. (2017), Go used less memory than C and C++, and Perl used a lot more memory than these three. C++ was found to be faster than Java and Python in studies by Aruoba and Fernandez-Villaverde (2014), Eichhorn et al. (2018) and Alomari et al. (2015). Couto et al. (2017) found out in their study that C was the fastest language followed by Java, and they left behind Perl and Go. C and C++ were faster languages. C was the fastest and used least memory with Go following in a study by Nanz and Furia (2015). Java is believed to have high memory consumptions, and this was supported by Prechelt (2000) and Cesarini, Pappalardo and Santoro (2008).

Although the current literature shows that C and C++ are very effective languages in terms of execution time and memory usage, this thesis has shown that that's not always the case; it has been quite the opposite in this thesis. Both of them had quite high execution times, as C wasn't the fastest language in any operation, and C++ was the fastest language in only 2 operations. C was slower than Python, Ruby, Go and Perl in three operations. C++ also lost in execution times to Python in five operations, to Ruby, Go and Perl in four operations, and to Java in three operations. It has been shown in current literature that Java consumes a lot of memory, and it was supported by this thesis, too; Java used the most memory in every operation.

The research questions for this thesis were:

1. Which programming languages have the shortest execution times on string operations?

2. Which programming languages use the least memory on string operations?

In a nutshell, there was no clear distinction on which language was the fastest or which used the least memory. However, it seems that Go is at least a real fast option, and

scripting languages (Python, Perl and Ruby) had quite solid results for execution times, although none of them stood out as a significantly fast language for string operations. The languages that used very little memory were C and C++, and Python used more memory than these two, but Python used additional memory to loading the string file in only one operation. Perl had also quite solid results for memory usages. In regular expressions, Perl and C++ had exceptionally great results in both execution times and memory usages.

In only two operations the differences between scripting and compiled languages could be concluded; in reverse operation, scripting languages had better execution times, and in replace operation, scripting languages had lower memory usages. The following will discuss the results in more detail.

Naturally, there were some diversity between the average execution times and memory usages between the operations. Sort, replace, and reverse were found to be more demanding operations in both execution time and memory usage, and regular expressions were clearly slow operations, but didn't use that much memory. These more demanding operations will now be called heavy operations, and the other ones will be called light operations.

There were 5 languages in which sorting a string was implemented: Ruby, Python, Go, Swift, and C++. Sorting seemed to be more demanding for scripting languages (Python and Ruby), as the maximum RSS usage was clearly the highest for Ruby (6.776 GB on 100m string), and Python was in the middle (1.147 GB). It is also worth noting that this was the only operation in which Python used any additional memory to loading strings. C++ didn't use any additional memory, despite being relatively slow on the task; the execution time for C++ was 41.4 seconds for 100m strings on average. Swift was distinctly the slowest with execution time of 210 seconds, and Ruby was the second slowest with 57 seconds. Python's execution time was 7.9 seconds, and Go's was 7.8 seconds, so those two were very even. No conclusions can be made on whether scripting or compiled languages were faster on average.

Reversing a string was implemented in Ruby, Perl, Python, Java, C++, and Swift. In this operation, scripting languages turned out to perform better than compiled languages. As for execution times, Ruby was the fastest (70ms on 100m string), followed by Perl (78ms), and Python (100ms), so every scripting language was faster than any compiled language. Java is a hybrid language, and it was also faster than the compiled languages, C++ and Swift, with an execution time of 0.26 seconds. Swift was very slow, as it took 11.2 seconds on average to reverse a 100m string, and for C++ it was 1.06 seconds, meaning that Swift was over ten times slower than the second slowest language. In memory usages C++, Python, and Swift didn't use any additional memory to loading the string file. Java used 1.145 GB memory, Ruby used 212 MB, and Perl used 201 MB.

Replacing substrings was implemented in Java, Swift, Go, Ruby, and Python. Scripting languages seemed to do better in this test, for maximum RSS usages at least: Python didn't use any additional memory to loading a string file, so Python used the least memory, and Ruby used the second least. Java used the most memory, followed by Swift and Go. In execution times though, Go was the fastest, followed by Python and Ruby. Swift was by far the slowest with an execution time of 21.9 seconds on 100m string, and Java was the second slowest with 1 second.

Regular expressions were implemented on all languages except for C. However, not much conclusions can be made between scripting and compiled languages from the

results for execution times. Perl was the fastest in every regex operation and C++ was only marginally slower, and the gap between these two and the other languages is so big that there is nothing to analyze on Perl and C++. As for the other languages, regex 4 was the only regex operation in which there seemed to be some distinction between scripting and compiled languages; Python and Ruby were the two fastest languages after Perl and C++ in regex 4. In other regex operations the results were very tied between these groups. Go's results for regex operations were quite peculiar; in regex 1 and 2, Go was the best language (excluding Perl and C++), but in regex 3, 4, and 5, its execution times increased greatly.

There were much less variation in the results for light operations. Java, Python, Perl, and Go were the slowest languages in string concatenation, possibly because strings in those are immutable. In string concatenation in Ruby, results for execution time for 10m and 100m strings were very peculiar, as the execution time increased rapidly from 1m to 10m string, and dropped again from 10m to 100m string. This was probably caused by Ruby's Time-library used for execution time. Appendix B shows the execution times with more string sizes.

String equality is another light operation, but Swift's results for execution times were very slow. It seems that Swift's elementsEqual-method is designed for more general comparisons rather than simple string comparisons. It took 23.5 seconds on average to compare two 100m strings in Swift, whereas it took only 9.1 milliseconds in C++, which was the second slowest language in equality comparison. However, when elementsEqual-method was replaced with ==-operator, the execution time dropped to 9 milliseconds. Regardless of how tremendously slow elementsEqual-method was, it didn't use any additional memory to loading the string file.

In string duplication, the languages had to create a copy of the string. C and C++ were the only language that required additional memory in execution, and they were also clearly the slowest languages in the operation. The execution time for 100m string was 49 milliseconds in C, and 43 milliseconds in C++, whereas in other languages the slowest was Python, in which the execution time was 0.009 milliseconds. This could be because C and C++ created deep copies of the strings, but other languages created shallow copies.

Python and C had the least variation on max RSS usages overall; only one operation in both of them used more memory than what was required for loading the string file. However, it should be noted that fewer operations were implemented in C. C++ used additional memory in two operations. The rest of the languages had more variation in their memory usages. Strings in Java, Go, Perl, and Python are immutable, which could cause some of the peaks in their memory usages; for example, when reversing a string in a language that has immutable strings, the language creates a new string, which then is reversed. However, it seems that this is not the case in Python, because sorting a string was the only operation in which Python used any additional memory. The most memory was used by Java and Swift.

6. Conclusion

This paper has presented the execution times and memory usages on various string operations across eight programming languages (Java, Swift, Go, Python, C++, Ruby, Perl, and C). While there was no sole winner for neither execution times nor memory usages, some languages performed better on overall than others.

Looking at the big picture, Go seemed to be the most effective language in execution times; it was the fastest language on five operations, and in regular expressions, it was the fastest after C++ and Perl in two out of five operations. C++ and Perl were overwhelmingly effective languages in regular expressions in execution times - in memory usages, there weren't much variation across languages.

Swift was remarkably slow; it was the second fastest language in two operations, but it was the slowest in seven operations. The reason for Swift's poor results could be caused by Swift's Linux version. Swift used 96 MB memory in only loading the smallest string size (ten thousand characters), while the least memory was used by C (1.3 MB) and Go (1.9 MB). In loading the biggest string size (100 million characters), Java used 754 MB memory, which was the most, and Swift used the second most with 389 MB, so the gap between these two is quite remarkable. Java used the most memory in every operation it was implemented in when the operation was performed on the biggest string size.

In two operations scripting languages had better results than compiled languages; in reverse operation, scripting languages had better execution times, and in replace operation, scripting languages had lower memory usages. There was also no clear distinction among scripting languages. In compiled languages, Go was faster and used less memory than Swift and Java in overall.

One possible limitation is that some of the solutions implemented for programming languages might not be the best solutions there are (e.g. in Python, is *if 'abc' in literals* truly the best solution for checking whether string includes a substring?), although built-in functions/methods were only used (except for some exceptions, which are particularized). There might be some cases where a language could have had better results if it was implemented otherwise; this goes for both execution times and memory usages. For example, loading the string file might be implemented in a way that causes increase in memory usage. However, it could be quite difficult to implement the best solutions only, because the total number of languages and operations implemented in them is so high.

As for future research, a clear limitation in this thesis was that C# had to be excluded because it is not supported by Linux-based operating systems; in future research, the performance of string operations on C# should be tested. What should be studied in the future is testing with other hardware, too; for example, with different operating systems, hard drive sizes, or drive types. Related to this, Swift should be tested on macOS rather than Linux-based operating systems. Also, there were some operations that weren't implemented (for example, sorting in C), and in a sense, this is a limitation.

References

Alomari, Z., El Halimi, O., Sivaprasad, K. & Pandit, C. (2015). Comparative Studies of Six Programming Languages. *ArXiv*.

Aruoba, S.B. & Fernandez-Villaverde, J. (2014). A Comparison of Programming Languages in Economics. *Journal of Economic Dynamics and Control, 58,* 265-273. doi: 10.3386/w20263

Bouckaert, S., Gerwen, J.V., & Moerman, I. (2011). Benchmarking computers and computer networks.

Busbee, K.L. (2009). *Programming Fundamentals: A Modular Structured Approach Using C++*. Retrieved from OpenStax College.

Cesarini, F., Pappalardo, V. & Santoro, C. (2008). A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol, *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, 29-40. doi: 10.1145/1411273.1411279

Character set of C. (n.d.). Retrieved September 3, 2019, from http://aboutc.weebly.com/c-character-set.html

class Encoding. (n.d.). Retrieved January 14, 2019 from https://docs.ruby-lang.org/en/2.1.0/Encoding.html

const type qualifier. (2018). Retrieved January 15, 2019 from https://en.cppreference.com/w/c/language/const

Cooper, P. (2009). *Beginning Ruby: From Novice to Professional, Second Edition* (pp. 39-41). New York, NY: Springer-Verlag New York.

Couto, M., Pereira, R., Ribeiro, F., Rua, R. & Saraiva, J. (2017). Towards a Green Ranking for Programming Languages, *Proceedings of the 21st Brazilian Symposium on Programming Languages*. doi: 10.1145/3125374.3125382

cplusplus. (n.d.-a). string. Retrieved January 11, 2019, from http://www.cplusplus.com/reference/string/string/

cplusplus. (n.d.-b). Variables and types. Retrieved January 11, 2019, from http://www.cplusplus.com/doc/tutorial/variables/

cplusplus. (n.d.-c). wstring. Retrieved January 11, 2019, from http://www.cplusplus.com/reference/string/wstring/

Eichhorn, H., Cano, J.L., McLean, F. & Anderl, R. (2018). A comparative study of programming languages for next-generation astrodynamics systems, *CEAS Space Journal*, 10(1), 115-123. doi: https://doi.org/10.1007/s12567-017-0170-8

Fourment, M. & Gillings, M.R. (2008). A comparison of common programming languages used in bioinformatics. Retrieved December 5, 2018, from https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-82

GitHub. (n.d.). The State of the Octoverse. Retrieved November 30, 2019, from https://octoverse.github.com/

Gustedt, J., Jeannot, E. & Quinson, M. (2009). Experimental Methodologies for Large-Scale Systems: a Survey, *Parallel Processing Letters, World Scientific Publishing,* 19(3), 399-418.

IBM Knowledge Center. (n.d.). Memory usage determination with the ps command. Retrieved October 7, 2019, from https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/performance/mem_usage_determine_ps.html

Ilseman, M. (2019). UTF-8 String [Blog post]. Retrieved September 3, 2019, from https://swift.org/blog/utf8-string/

Jay, G., Hale, J.E., Smith, R.K., Hale, D., Kraft, N.A. & Ward, C. (2009). Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship, *Journal of Software Engineering and Applications,* Volume 2, 137-143. doi: 10.4236/jsea.2009.23020

Jones, T. C. (1978). Measuring programming quality and productivity, *IBM Systems Journal,* 17(1), 39-63. doi: 10.1147/sj.171.0039

Landman, D., Serebrenik, A. & Vinju, J. (2014). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods, *2014 IEEE International Conference on Software Maintenance and Evolution.* 221-230. doi: 10.1109/ICSME.2014.44

Landman, D., Serebrenik, A., Bouwers, E. & Vinju, J.J. (2015). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions, *Journal of Software: Evolution and Process,* 28(7), 589-618. doi: https://doi.org/10.1002/smr.1760

Lerner, R. (2002). *Core Perl, Third Edition* (p. 22). New Jersey, NJ: Prentice Hall PTR.

Nanz, S. & Furia, C.A. (2015). A comparative study of programming languages in Rosetta Code, *Proceedings of the 37th International Conference on Software Engineering,* Volume 1, 778-788.

Neuman, W.L. (2011). *Social Research Methods: Qualitative and Quantitative Approaches, 7th Edition.* England, GB-ENG: Pearson.

Normand, E. (2019). The Legend of Long JVM Startup Times [Blog post]. Retrieved February 10, 2019, from https://purelyfunctional.tv/article/the-legend-of-long-jvm-startup-times/

Oracle. (n.d.-a). Class Charset. Retrieved September 3, 2019, from https://docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html

Oracle. (n.d.-b). Class String. Retrieved January 14, 2019 from https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Oracle. (n.d.-c). Java Virtual Machine Technology Overview. Retrieved January 11, 2019, from https://docs.oracle.com/javase/10/vm/java-virtual-machine-technology-overview.htm#JSJVM-GUID-982B244A-9B01-479A-8651-CB6475019281

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P. & Saraiva, J. (2017). Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 256-267. doi: 10.1145/3136014.3136031

Pike, R. (2013a). Arrays, slices (and strings): The mechanics of 'append' [Blog post]. Retrieved January 20, 2019, from https://blog.golang.org/slices

Pike, R. (2013b). Strings, bytes, runes and characters in Go [Blog post]. Retrieved September 3, 2019, from https://blog.golang.org/strings

Prechelt, L. (2000). An Empirical Comparison of Seven Programming Languages, *IEEE Computer, 33(10)*, 23-29. doi: 10.1109/2.876288

Python Reference Manual. (n.d.). Objects, values and types. Retrieved January 11, 2019 from https://docs.python.org/2.0/ref/objects.html

Rao, S.N. (2015). Why Java is both compiled and interpreted language? [Blog post]. Retrieved September 16, 2019 from: https://way2java.com/oops-concepts/why-java-is-both-compiled-and-interpreted-language/

Ruby user's guide. (n.d.). Class constants. Retrieved January 14, 2019, from https://ruby-doc.org/docs/ruby-doc-bundle/UsersGuide/rg/constants.html

Rosetta Code. (n.d.) Retrieved December 14, 2018, from http://rosettacode.org/wiki/Rosetta_Code

Spinellis, D., Karakoidas, V. & Louridas, P. (2012). Comparative Language Fuzz Testing: Programming Languages vs. Fat Fingers, Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools, 25-34. doi: 10.1145/2414721.2414727

Stack Overflow. (2018). Developer Survey Results 2018. Retrieved December 14, 2018, from https://insights.stackoverflow.com/survey/2018/

Ramanathan, N. (2017). Strings [Blog post]. Retrieved September 3, 2019, from: https://golangbot.com/strings/

Stroustrup, B. (2013). *The C++ Programming Language, Fourth Edition* (p. 1038). New Jersey, NJ: Pearson Inc.

Swift. (n.d.). Strings and Characters. Retrieved January 10, 2019, from https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html

The Python Wiki. (n.d.). DefaultEncoding. Retrieved September 3, 2019, from https://wiki.python.org/moin/DefaultEncoding

Tichy, W.F. (1998). Should Computer Scientists Experiment More?, *Journal Computer*, 31(5), 32-40. doi: 10.1109/2.675631

Vahtera, P. (2003). *Mikro-ohjaimen ohjelmointi C-kielellä* (p. 180). Porvoo, FI: WS Bookwell Oy.

Wainwright, P. (2005). *Pro Perl* (p. 959). New York, NY: Springer-Verlag New York.

Wall, L., Christiansen, T. & Orwant, J. (2000). *Programming Perl, Third Edition* (p. 842). Sebastopol, CA: O'Reilly & Associates, Inc.

W3C. (2000). Document Object Model (DOM) Level 1 Specification (Second Edition). Retrieved January 10, 2019 from: https://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/DOM.pdf

Appendix A. Results for all tests

A.1 Loading string file

Table 4. Execution times (in milliseconds) for loading string files.

Language	**10k**	**100k**	**1m**	**10m**	**100m**
Python	0.099628	0.177376	1.045463	9.149566	91.472783
C	0.00005	0.00013	0.000598	0.003999	0.037693
C++	0.777178	6.572439	62.216575	621.551147	6,182.25729
Java	0.738844	5.262166	23.805246	77.375556	495.880924
Perl	0.026958	0.088201	0.701015	5.403695	51.728222
Ruby	0.02428	0.069382	0.52322	4.580577	44.983452
Go	0.082493	0.240270	0.974235	8.221429	76.279186
Swift	0.643466	0.786504	2.396459	18.952958	182.111185

Table 5. Maximum RSS usages (in megabytes) for loading string files.

Language	**10k**	**100k**	**1m**	**10m**	**100m**
Python	8.525	8.630	10.304	27.993	203.554
C	1.269	1.350	2.239	11.029	98.914
C++	3.212	3.309	4.924	18.320	125.985
Java	25.623	28.023	34.518	113.855	753.395
Perl	5.849	5.948	6.874	15.668	103.560
Ruby	17.4	17.485	18.371	27.162	115.051
Go	1.890	1.855	3.667	22.149	203.710
Swift	96.011	96.257	98.816	125.256	388.696

A.2 Python

Table 6. Execution times (in milliseconds) for Python operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	0.001314	0.004370	0.355015	4.253683	47.214322
Replace	0.047951	0.412278	3.873708	39.367723	394.965672
Equals	0.000484	0.00062	0.001009	0.001428	0.001761
Uppercase	0.00645	0.052977	0.774305	7.90683	80.512569
First index of substring	0.001404	0.001304	0.002339	0.003564	0.003233
Duplicate	0.00216	0.002384	0.004482	0.007765	0.009346
Reverse	0.007844	0.067379	0.938077	9.652154	99.844880
Sort	0.726311	7.228434	73.719323	761.514282	7,929.862432
Regex 1	0.196266	0.923684	8.244894	80.272629	798.029141
Regex 2	0.165799	0.66144	5.481224	53.741951	532.754371
Regex 3	0.388896	2.660467	25.314102	252.637553	2,517.033820
Regex 4	0.289936	1.433284	12.658384	126.215141	1,244.827597
Regex 5	0.695474	5.65006	54.76459	546.781504	5,441.255755

Table 7. Maximum RSS usages (in megabytes) for Python operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	8.779	8.893	10.647	28.066	203.874
Replace	8.594	8.819	10.307	28.000	203.552
Equals	8.775	8.930	10.634	28.045	203.889
Uppercase	8.536	8.759	10.3012	27.985	203.550
First index of substring	8.535	8.636	10.313	27.985	203.557
Duplicate	8.528	8.620	10.307	28.001	203.554
Reverse	8.542	8.762	10.414	27.854	203.669
Sort	8.616	9.505	20.349	128.289	1,147.157
Regex 1	8.984	9.106	10.663	28.088	203.904
Regex 2	8.332	8.949	10.672	28.099	203.906
Regex 3	8.847	8.980	10.658	28.086	203.915
Regex 4	8.976	9.105	10.668	28.093	203.923
Regex 5	8.814	8.935	10.649	28.053	203.892

A.3 C

Table 8. Execution times (in milliseconds) for C operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	0.00276	0.00556	0.056	0.54673	5.38508
Equals	0.000004	0.000003	0.000064	0.000918	0.009299
Uppercase	0.03124	0.1947	1.96438	16.78364	167.57057
First index of substring	0.00189	0.00208	0.00219	0.00296	0.00305
Duplicate	0.0059	0.06405	0.43345	4.96427	48.99561

Table 9. Maximum RSS usages (in megabytes) for C operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	1.282	1.366	2.248	11.039	98.935
Equals	1.330	1.501	3.265	20.841	196.613
Uppercase	1.270	1.355	2.256	11.044	98.933
First index of substring	1.274	1.350	2.250	11.026	98.927
Duplicate	1.288	1.457	3.208	20.793	196.580

A.4 C++

Table 10. Execution times (in milliseconds) for C++ operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	0.001573	0.001890	0.003174	0.003788	0.003824
Equals	0.001555	0.005364	0.072012	0.922625	9.126494
Uppercase	0.05454	0.484182	4.681662	46.644758	465.165919
Duplicate	0.002021	0.005974	0.359146	4.38781	42.691904
Reverse	0.108624	1.094504	10.639584	106.368017	1063.94603
Sort	1.974374	24.917114	298.790792	3,517.208525	41,368.53114
First index of a substring	0.004208	0.004220	0.006240	0.007392	0.006867
Regex 1	0.101013	0.089078	0.67535	0.081559	0.134780
Regex 2	0.044977	0.151356	0.071106	0.073972	0.051161
Regex 3	0.061957	0.071412	0.113078	0.167207	0.094620
Regex 4	0.072948	0.115108	0.076647	0.094849	0.085732
Regex 5	0.055937	0.078333	0.121770	0.182134	0.100893

Table 11. Maximum RSS usages (in megabytes) for C++ operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	3.216	3.315	4.942	18.332	125.992
Equals	3.240	3.413	5.938	32.195	296.093
Uppercase	3.220	3.424	4.981	18.369	126.025
Duplicate	3.216	3.312	4.946	22.436	198.422
Reverse	3.211	3.315	4.936	18.324	125.994
Sort	3.215	3.329	4.936	18.324	125.994
First index of a substring	3.284	3.372	4.944	18.324	125.991
Regex 1	3.518	3.611	5.135	18.518	126.191
Regex 2	3.522	3.617	5.128	18.519	126.185
Regex 3	3.523	3.632	5.131	18.517	126.193
Regex 4	3.520	3.628	5.133	18.524	126.186
Regex 5	3.518	3.622	5.127	18.522	126.181

A.5 Java

Table 12. Execution times (in milliseconds) for Java operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	0.01745	0.123431	1.037697	9.976855	98.238553
Replace	2.524518	9.696057	24.204217	116.63562	992.616618
Equals	0.002454	0.003905	0.003952	0.003768	0.003823
Uppercase	1.372016	9.420854	19.060621	82.262429	715.652189
First index of substring	0.00752	0.009915	0.011307	0.01422	0.010419
Duplicate	0.003715	0.005274	0.005386	0.006083	0.006406
Reverse	0.521287	5.525425	10.377626	33.295755	255.977597
Regex 1	2.518569	8.231486	31.571341	131.624146	1,084.98861
Regex 2	1.596917	5.286252	13.933691	89.497882	439.033592
Regex 3	2.902747	10.206512	35.014569	207.135255	1,862.306059
Regex 4	3.442515	12.183267	49.270333	243.553902	2,061.342968
Regex 5	3.831162	13.912213	49.74932	374.934954	3,666.892436

Table 13. Maximum RSS usages (in megabytes) for Java operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	25.648	28.209	36.513	133.463	949.499
Replace	26.091	30.007	42.601	173.214	1,057.731
Equals	25.644	28.195	36.443	133.354	949.568
Uppercase	26.109	28.894	38.463	153.154	1,145.564
First index of substring	25.621	28.002	34.481	113.811	753.394
Duplicate	25.648	28.022	34.536	113.832	753.403
Reverse	25.810	28.650	38.431	152.982	1,145.546
Regex 1	26.128	28.790	35.512	114.872	753.747
Regex 2	25.837	28.527	35.645	116.842	754.436
Regex 3	26.320	28.942	34.654	113.927	753.584
Regex 4	26.497	28.906	37.349	116.331	754.557
Regex 5	26.366	28.840	34.813	116.952	755.166

A.6 Perl

Table 14. Execution times (in milliseconds) for Perl operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	0.001774	0.039792	0.381601	4.450960	43.44
Equals	0.000982	0.001123	0.002165	0.002956	0.003953
Uppercase	0.007896	0.099435	0.974007	9.622917	94.806225
First index of substring	0.005167	0.074329	0.378370	7.952762	78.014257
Duplicate	0.001575	0.001512	0.002563	0.004761	0.004077
Reverse	0.000834	0.000904	0.001423	0.002487	0.00275
Regex 1	0.003083	0.002785	0.003805	0.006552	0.007508
Regex 2	0.003285	0.005417	0.006008	0.01018	0.009913
Regex 3	0.005329	0.006738	0.010662	0.017722	0.013988
Regex 4	0.007136	0.008941	0.007167	0.014048	0.013063
Regex 5	0.00618	0.009224	0.016742	0.028477	0.01682

Table 15. Maximum RSS usages (in megabytes) for Perl operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concatenate	5.852	5.977	7.666	25.430	200.978
Equals	5.840	5.987	7.663	25.431	200.967
Uppercase	5.838	5.976	7.663	25.436	200.979
First index of substring	5.845	5.987	7.668	25.437	200.973
Duplicate	5.846	5.951	6.875	15.666	103.554
Reverse	5.850	5.945	6.878	15.673	103.552
Regex 1	6.133	6.244	7.167	15.967	103.851
Regex 2	6.140	6.238	7.160	15.959	103.852
Regex 3	6.138	6.239	7.179	15.959	103.838
Regex 4	6.137	6.238	7.178	15.958	103.842
Regex 5	6.134	6.236	7.173	15.969	103.855

A.7 Ruby

Table 16. Execution times (in milliseconds) for Ruby operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat (time-clock)	0.002717	0.003709	0.003806	0.289909	0.189815
Concat (Process-clock)	0.002289	0.003372	0.003285	0.004395	0.00513
Replace	0.075015	0.722671	7.037939	70.742676	704.477261
Equals	0.001078	0.001028	0.001457	0.01022	0.004196
Uppercase	0.041494	0.426681	4.221021	42.840063	425.637103
First index of substring	0.002412	0.00236	0.003404	0.005484	0.005068
Duplicate	0.002129	0.002141	0.002741	0.005908	0.004533
Reverse	0.005415	0.071545	0.706630	7.370111	70.177689
Sort	4.330921	41.00965	427.539195	5,151.234331	57,010.85816
Regex 1	0.042577	0.357598	4.699143	36.180190	343.948202
Regex 2	0.0468	0.396645	5.184476	39.174612	375.227543
Regex 3	0.327028	3.148043	32.482678	310.557177	3,106.837626
Regex 4	0.188468	1.844108	19.244438	179.780084	1,809.142028
Regex 5	0.379926	3.627959	36.889105	358.531016	3,556.172481

Table 17. Maximum RSS usages (in megabytes) for Ruby operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	17.401	17.502	18.367	27.159	115.051
Replace	17.408	17.589	19.350	36.914	212.729
Equals	17.414	17.594	19.353	36.923	212.726
Uppercase	17.411	17.583	19.345	36.922	212.724
First index of substring	17.403	17.490	18.371	27.153	115.054
Duplicate	17.401	17.495	18.375	27.151	115.046
Reverse	17.405	17.579	19.349	36.927	212.729
Sort	17.896	24.100	84.558	730.622	6,775.849
Regex 1	17.464	17.543	18.485	28993	134.016
Regex 2	17.461	17.548	18.489	29.009	134.154
Regex 3	17.458	17.553	18.572	29.645	143.058
Regex 4	17.463	17.548	18.642	30.520	151.694
Regex 5	17.462	17.554	18.772	31.782	162.184

A.8 Go

Table 18. Execution times (in milliseconds) for Go operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	0.010175	0.051462	0.353178	1.294861	9.358545
Replace	0.042268	0.368519	3.354867	30.184500	290.237038
Equals	0.000169	0.000152	0.000163	0.000177	0.000181
Uppercase	0.073306	0.704424	5.805305	53.563662	523.626440
First index of substring	0.000659	0.000787	0.000690	0.000993	0.000806
Duplicate	0.000185	0.000149	0.000200	0.000251	0.000252
Sort	0.696396	6.187541	93.948468	799.360018	7,845.185754
Regex 1	0.111871	0.306094	2.493987	25.222674	260.177756
Regex 2	0.096922	0.342083	3.003489	29.097139	294.630616
Regex 3	0.773743	7.668254	75.945236	784.497655	7,513.534611
Regex 4	0.418946	4.755594	43.442788	443.512254	4,424.130852
Regex 5	0.909471	8.868731	84.295033	849.326940	8,428.638010

Table 19. Maximum RSS usages (in megabytes) for Go operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	1.893	1.965	4.689	22.567	203.730
Replace	1.866	2.029	5.896	32.407	304.645
Equals	1.896	1.977	4.690	23.268	203.721
Uppercase	1.860	1.955	4.714	22.784	203.742
First index of substring	1.836	1.868	3.676	22.156	203.704
Duplicate	1.888	1.848	3.677	22.151	203.709
Sort	1.770	3.398	20.046	183.520	1,816.775
Regex 1	2.084	2.271	4.389	22.695	240.408
Regex 2	2.070	2.264	4.384	22.645	240.491
Regex 3	2.074	2.285	4.502	22.701	251.823
Regex 4	2.080	2.311	4.665	22.728	266.582
Regex 5	2.093	2.308	4.591	22.545	259.120

A.9 Swift

Table 20. Execution times (in milliseconds) for Swift operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	0.00202	0.00203	0.00416	0.004	0.00462
Replace	2.480699	22.498182	216.580808	2,183.818428	21,866.630097
Equals (elementsEqual)	2.386845	23.570849	234.95634	2,352.729166	23,514.53078
Equals (with if-else)	0.00305	0.00656	0.06601	0.90868	9.01657
Uppercase	0.014576	0.131117	1.638104	17.080966	171.591638
Reverse	1.194376	11.234932	112.301227	1,119.999102	11,229.241265
Duplicate	0.000743	0.000753	0.001537	0.001544	0.001823
Sort	12.350322	142.518714	1,639.774702	19,082.065926	210,382.269161
Regex 1	3.09842	6.85402	44.99106	427.92528	4,244.90508
Regex 2	3.05565	6.74803	43.81262	416.30084	4,118.68323
Regex 3	3.24778	8.6681	63.10698	609.37921	6,048.97838
Regex 4	3.12021	7.36424	49.96678	478.27917	4,739.0154
Regex 5	3.33812	9.4577	71.1229	689.60137	6,850.54493

Table 21. Maximum RSS usages (in megabytes) for Swift operations.

Operation	**10k**	**100k**	**1m**	**10m**	**100m**
Concat	95.834	96.128	98.687	125.09	388.589
Replace	105.084	105.474	108.432	143.586	494.594
Equals (elementsEqual)	101.415	101.718	104.138	130.567	394.059
Equals (if-else)	101.273	101.599	104.136	130.531	393.998
Uppercase	96.881	97.146	99.740	126.137	389.623
Reverse	98.356	98.638	101.19	127.67	391.118
Duplicate	96.475	96.749	99.349	125.734	389.219
Sort	97.622	98.197	105.794	185.096	975.947
Regex 1	101.825	102.166	104.392	134.979	439.446
Regex 2	101.852	102.143	104.365	134.971	439.733
Regex 3	101.384	102.176	104.483	136.546	458.337
Regex 4	101.887	102.136	104.811	138.721	476.857
Regex 5	101.794	102.158	104.542	137.372	463.409

Appendix B. Results for string concatenation in Ruby

Table 22. Results for string concatenation in Ruby (when Ruby's Time-library was used). The number of string sizes were added in order to find out why the execution time suddenly grew for 10m string and dropped again for 100m string.

String size	Time (ms)
1m	0.00355
2m	0.004685
3m	0.004867
4m	0.005212
5m	0.004739
6m	0.005548
7m	0.004856
8m	0.005210
9m	0.286959
10m	0.288011
20m	0.005008
30m	0.004864
40m	0.191078
50m	0.193088
60m	0.193924
70m	0.189643
80m	0.191483
90m	0.199214
100m	0.194589

Appendix C. Execution times and memory usages as positions across languages

Table 23. Summary of execution times by positions on 100m string (1=lowest execution time). Cells with a grey color mean that the operation was not implemented.

Operation	**C**	**C++**	**Java**	**Swift**	**Go**	**Perl**	**Ruby**	**Python**
Concat	4	1	8	2	5	6	3	7
Uppercase	4	1	8	5	7	3	6	2
Equals	6	7	4	8	1	3	5	2
Duplication	8	7	4	2	1	3	6	5
Reverse		5	4	6		2	1	3
First index of substring	2	6	7		1	4	5	3
Replace			4	5	1		3	2
Sort		3		5	1		4	2
Regex 1		2	6	7	3	1	4	5
Regex 2		2	5	7	3	1	4	6
Regex 3		2	3	6	7	1	5	4
Regex 4		2	5	7	6	1	4	3
Regex 5		2	4	6	7	1	3	5

Table 24. Summary of maximum RSS usages by positions on 100m string (1=least memory used). Cells with a grey color mean that the operation was not implemented, and underlined numbers mean that the operation used additional memory to loading the string.

Operation	**C**	**C++**	**Java**	**Swift**	**Go**	**Perl**	**Ruby**	**Python**
Concat	1	3	8	7	5	4	2	6
Uppercase	1	2	8	7	5	3	6	4
Equals	1	6	8	7	3	2	5	4
Duplication	3	4	8	7	5	1	2	6
Reverse		1	6	5		2	4	3
First index of substring	1	4	7		6	2	3	5
Replace			5	4	3		2	1
Sort		1		2	4		5	3
Regex 1		2	7	6	5	1	3	4
Regex 2		2	7	6	5	1	3	4
Regex 3		2	7	6	5	1	3	4
Regex 4		2	7	6	5	1	3	4
Regex 5		2	7	6	5	1	3	4

Appendix D. Libraries and methods used for calculating the execution times

Table 25. Libraries and methods used in each language for calculating the execution times for operations.

Language	**Library**	**Method**
C	time.h	clock
C++	chrono	high_resolution_ clock::now
Java	System	nanoTime
Swift	time.h	clock (C-type library)
Go	time	time.Now
Perl	Time::HiRes qw(time)	time
Ruby	Time	now
Python	time	time

Appendix E. Source codes

All test codes are available at GitHub: https://github.com/npelkone/string-comparisons